Chương 4:
Kỹ thuật viết mã nguồn hiệu quả

Nội dung

1. Các kỹ thuật viết mã nguồn hiệu quả
2. Những nguyên tắc cơ bản trong việc tăng hiệu quả viết mã nguồn
3. Tối ưu hóa mã nguồn C/C++

Chương trình hiệu quả

- 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 liên tiếp kể từ m

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

```
void main(){
  long n, m, sum;
  cin << n;
  cin << m;
  sum = (m + m+ n) * n / 2;
  cout << "Sum = " << sum;
}
```


Dùng chỉ thị chương trình dịch

- Một số compilers 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?

Viết chương trình hiệu quả

- Xác định nguồn gây kém hiệu quả
  - Dư thừa tính toán - redundant computation
  - Chủ yếu
    - Trong các procedure
    - 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;
   //
   …
}
```


Hàm nội tuyến (inline functions)

Điều gì xảy ra khi một hàm được gọi?

- CPU sẽ **lưu địa chỉ bộ nhớ** của dòng lệnh hiện tại mà nó đang thực thi (để biết nơi sẽ quay lại sau lời gọi hàm),
- **Sao chép các đối số** của hàm trên ngăn xếp (stack)
- **Chuyển hướng điều khiển** sang hàm đã chỉ định.
  - CPU sau đó thực thi mã bên trong hàm
  - Lưu trữ giá trị trả về của hàm trong một vùng nhớ/thanh ghi
- Trả lại quyền điều khiển cho vị trí lời gọi hàm

→ Điều này sẽ tạo ra một lượng chi phí hoạt động nhất định (overhead) so với việc thực thi mã trực tiếp (không sử dụng hàm).

Hàm nội tuyến (inline functions)

- Đối với các **hàm lớn** hoặc các **tác vụ phức tạp** , tổng chi phí overhead của lệnh gọi hàm thường không đáng kể so với lượng thời gian mà hàm mất để chạy.
- Tuy nhiên, đối với các hàm nhỏ, thường xuyên được sử dụng, thời gian cần thiết để thực hiện lệnh gọi hàm thường nhiều hơn rất nhiều so với thời gian cần thiết để thực thi mã của hàm.
- **Inline functions (hàm nội tuyến)** là một loại hàm trong ngôn ngữ lập trình C++.
  - Từ khoá **inline** được sử dụng để **đề nghị (không phải là bắt buộc)** compiler thực hiện **inline expansion (khai triển nội tuyến)**
    - **chèn code của hàm đó tại địa chỉ mà nó được gọi.**

Inline functions

```
#include <iostream>
#include <cmath>
using namespace std;
inline double hypothenuse (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 << hypothenuse (k, m) << endl;
    cout << sqrt (k * k + m * m) << endl;
    return 0;
}
```

```
10.8167
10.8167
```

Inline functions

```
#include <iostream>
using namespace std;
inline int max(int a, int b){
  return a > b ? a : b;
}
int main() {
  cout << max(3, 6) << '\n';
  cout << max(6, 3) << '\n';
  return 0;
}
```

Khi chương trình trên được biên dịch, mã máy được tạo ra tương tự như hàm **main()** bên dưới:

```
int main(){
  cout << (3 > 6 ? 3 : 6) << '\n';
  cout << (6 > 3 ? 6 : 3) << '\n';
  return 0;
}
```


Inline functions

Trình biên dịch **có thể không thực hiện nội tuyến** trong các trường hợp như:

- Hàm chứa vòng lặp (for, while, do-while).
- Hàm chứa các biến tĩnh.
- Hàm đệ quy.
- Hàm chứa câu lệnh switch hoặc goto.

Inline functions

- **Ưu điểm:**

– Tiết kiệm chi phí gọi hàm.

– Tiết kiệm chi phí của các biến trên ngăn xếp khi hàm được gọi.

– Tiết kiệm chi phí cuộc gọi trả về từ một hàm.

- **Nhược điểm:**

– Tăng kích thước file thực thi do sự trùng lặp của cùng một mã.

– Khi được sử dụng trong file tiêu đề (*.h), nó làm cho file tiêu đề của bạn lớn hơn.

– Hàm nội tuyến có thể không hữu ích cho nhiều hệ thống nhúng. Vì trong các hệ thống nhúng, kích thước mã quan trọng hơn tốc độ.

Macros

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

- Cách hoạt động:
  - Macro là một tính năng của tiền xử lý (preprocessor), nghĩa là chúng được thay thế trực tiếp vào mã trước khi quá trình biên dịch thực sự bắt đầu.
- Không kiểm tra kiểu dữ liệu:
  - Macro không kiểm tra kiểu dữ liệu, dẫn đến khả năng gây ra lỗi khi sử dụng với các kiểu dữ liệu khác nhau hoặc khi có vấn đề về toán tử.
- Không có khả năng gỡ lỗi:
  - Vì macro chỉ là sự thay thế trực tiếp của mã, việc gỡ lỗi có thể trở nên khó khăn. Lỗi do macro gây ra thường khó phát hiện và sửa chữa hơn.
- Phạm vi:
  - Macro không tuân theo quy tắc phạm vi, chúng chỉ là sự thay thế văn bản và có thể gây ra xung đột tên (naming conflicts).
- Cú pháp:
  - Macro thường không rõ ràng như inline function, và các lỗi tiềm ẩn có thể xảy ra do cách các toán tử được đánh giá.

Cách xử lý của Macro

- Tiền xử lý (Preprocessing):
  - Macro được xử lý bởi trình tiền xử lý (preprocessor) trước khi quá trình biên dịch thực sự bắt đầu. Trình tiền xử lý thực hiện các công việc như thay thế macro, xử lý chỉ thị #include, và các chỉ thị tiền xử lý khác.
- Thay thế văn bản đơn giản:
  - Khi định nghĩa một macro bằng #define, trình tiền xử lý sẽ tìm và thay thế tất cả các xuất hiện của macro trong mã nguồn bằng nội dung đã định nghĩa. Quá trình này là thay thế văn bản thuần túy, không có hiểu biết về ngữ nghĩa của mã.

- □ Hiệu ứng phụ không mong muốn. Ví dụ:
  - `#define SQUARE(x) x * x`
    - Gọi SQUARE(1 + 2) sẽ thành 1 + 2 * 1 + 2, kết quả là 5 thay vì 9

Inline function vs. Macro

Đặc điểm	Inline Function	Macro
Cách xử lý	Trình biên dịch xử lý	Trình tiền xử lý xử lý
Kiểm tra kiểu dữ liệu	Có (ở thời gian biên dịch)	Không
Phạm vi	Tuân theo quy tắc phạm vi	Không tuân theo, dễ gây lỗi xung đột
Hiệu suất	Tương đương macro	Tương đương inline function
Khả năng gỡ lỗi	Dễ dàng, hỗ trợ bởi trình biên dịch	Khó khăn
Bảo trì	Dễ bảo trì hơn	Khó bảo trì hơn

Khi nào nên dùng?

- Cả inline function và macro đều được sử dụng để tối ưu hóa hiệu suất bằng cách tránh gọi hàm thông thường
  - Sử dụng inline function khi bạn muốn hiệu suất tương tự macro nhưng với tính an toàn và dễ bảo trì của một hàm thông thường.
  - Sử dụng macro cho các đoạn mã đơn giản

→ Kết luận: inline function thường an toàn và đáng tin cậy hơn macro trong hầu hết các tình huống.

Ví dụ Macros

```
#include <iostream>
#define for(i,a,b) for(int i = a; i <= b; i++)
using namespace std;
int main(){
    int n = 5, sum = 0;
    for(i,0,n) sum += i;
    cout << sum << endl;
    return 0;
}
```

Kết quả: 15

Ví dụ Macros

Dễ gây nhầm lẫn

```
#include <iostream>
#define expr 2 + 5
using namespace std;
int main(){
    cout << 3 * expr;
    return 0;
}
```

Kết quả: 11 = 3 * 2 + 5 = 11

```
#include <iostream>
#define expr (2 + 5)
using namespace std;
int main(){
    cout << 3 * expr;
    return 0;
}
```

Kết quả: 21 = 3 * (2+5)

Biến cục bộ thông thường

- Biến cục bộ (trong hàm) thông thường (không được khai báo với từ khoá static)
  - Các biến khai báo trong chương trình con được cấp phát bộ nhớ khi chương trình con được gọi và sẽ bị loại bỏ khi kết thúc chương trình con.
  - Khi gọi lại chương trình con, các biến cục bộ lại được cấp phát và khởi tạo lại.

```
#include <iostream>
using namespace std;
void counter() {
    int count = 0;  //Biến cục bộ thông thường
    count++;
    cout << "Count: " << count << endl;
}

int main() {
    counter();  // In ra: Count: 1
    counter();  // In ra: Count: 1
    counter();  // In ra: Count: 1
    return 0;
}
```

Biến cục bộ (trong hàm) với static (biến tĩnh cục bộ, static variable)

- Khi một biến cục bộ trong hàm được khai báo với từ khóa static, biến đó sẽ chỉ được khởi tạo một lần duy nhất trong suốt thời gian chạy của chương trình, thay vì được khởi tạo lại mỗi khi hàm được gọi. Sau lần khởi tạo đầu tiên, giá trị của biến static sẽ được giữ nguyên giữa các lần gọi hàm.
  - Thời gian sống (lifetime): Biến static trong hàm có thời gian sống từ lúc chương trình bắt đầu cho đến khi chương trình kết thúc, nhưng phạm vi sử dụng chỉ giới hạn trong hàm đó.
  - Khởi tạo một lần: Biến chỉ được khởi tạo một lần duy nhất và giữ giá trị giữa các lần gọi hàm.

Biến tĩnh cục bộ (static variables)

- Trong ví dụ này, giá trị của biến count được giữ nguyên giữa các lần gọi hàm counter()

```
#include <iostream>
using namespace std;
void counter() {
    static int count = 0;//Chỉ được khởi tạo một lần
    count++;
    cout << "Count: " << count << endl;
}

int main() {
    counter();  // In ra: Count: 1
    counter();  // In ra: Count: 2
    counter();  // In ra: Count: 3
    return 0;
}
```

So sánh biến tĩnh cục bộ và biến toàn cục thông thường

- Dùng biến cục bộ với static thay vì biến toàn cục?
  - Ưu điểm của 1 biến cục bộ với static: Phạm vi hẹp, dễ quản lý; Tăng hiệu suất; và Bảo vệ dữ liệu

Đặc điểm	Biến cục bộ với `static`	Biến toàn cục thông thường
Phạm vi	Chỉ trong hàm chứa nó	Toàn bộ chương trình, có thể truy cập từ mọi nơi
Thời gian sống	Từ khi chương trình bắt đầu đến khi kết thúc	Từ khi chương trình bắt đầu đến khi kết thúc
Khả năng truy cập	Chỉ trong hàm, giúp bảo vệ dữ liệu	Truy cập từ mọi nơi trong chương trình
Giữ giá trị giữa các lần gọi	Có (giữ giá trị giữa các lần gọi hàm)	Có (giữ giá trị suốt chương trình)
Tránh xung đột tên	Có (vì chỉ có phạm vi trong hàm)	Không (dễ gây xung đột tên trong chương trình lớn)
Dữ liệu chia sẻ giữa các hàm	Không dễ chia sẻ (vì phạm vi hẹp)	Dễ chia sẻ dữ liệu giữa các hàm hoặc file khác

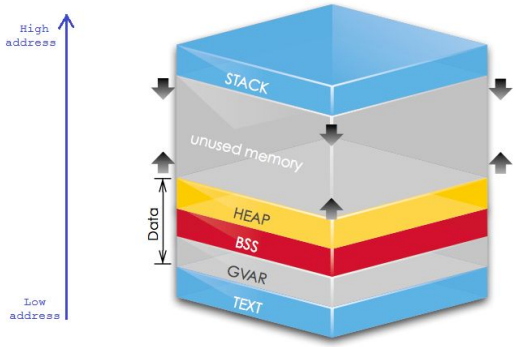
Phạm vi

- So sánh phạm vi của biến cục bộ với static và biến toàn cục

```
#include <iostream>
using namespace std;

tnt dem = 4;//Biến toàn cục, được truy cập trong toàn bộ chương trình
void counter() {
    static int count = 0;
    count++;
    cout << "Count: " << count << endl;
}

int. main() {
    counter();  // In ra: Count: 1
    counter();  // In ra: Count: 2
    counter();  // In ra: Count: 3

    cout << ”Dem: " <<  dem << endl;
    cout << "Count: " <<  count << endl;//Lỗi: count là cục bộ của hàm counter
                                        -> chỉ được truy cập trong counter

    return 0;
}
```

Static, Stack, và heap

- Khi thực hiện, vùng dữ liệu data segment của một chương trình được chia làm 3 phần:
  - static, stack, và heap data.
- Static:
  - global hay static variables
- Stack data:
  - các biến cục bộ của chương trình con
- Heap data:
  - Dữ liệu được cấp phát động
  - Dữ liệu này sẽ còn cho đến khi ta giải phóng hoặc khi kết thúc chương trình.



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;
```

```
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, found, size;
// Gán giá trị cho s, searchValue
…
size = strlen(s);
pos = 0;
while (pos < size) && (s[pos] != searchValue){pos++;}

if  (pos >= size) found =0
else found = 1;
```

⇒ **Phải thực hiện kiểm tra pos < size ở mỗi lần lặp**

Dùng "lính canh" ….

- Ý tưởng chung
  - **Đặt giá trị cần tìm (`searchValue`) vào cuối xâu**
  - Luôn đảm bảo tìm thấy giá trị cần tìm.
  - Nhưng nếu vị trí >= size nghĩa là không tìm thấy!

```
size = strlen(s);
strcat(s, searchValue);
pos = 0;
while ( s[pos] != searchValue){pos++;}

if  (pos >= size) found = 0
else found = 1;
⇒ Không phải thực hiện kiểm tra pos < size ở mỗi lần lặp
```

Có thể làm tương tự với mảng, danh sách …

Dịch chuyển những biểu thức bất biến ra khỏi vòng lặp

- Đừng 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;
```

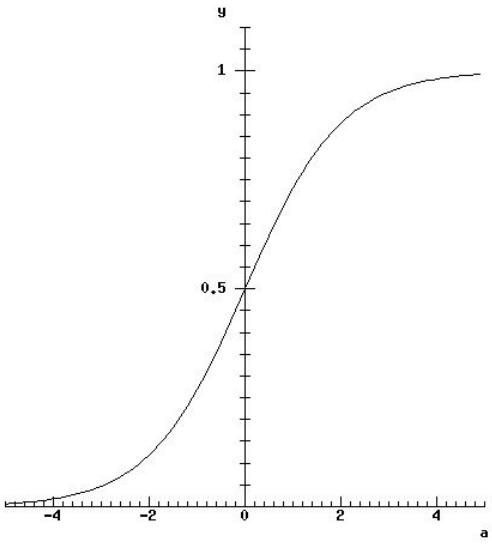
```
i = j;
sum += q*i -i*7;
i ++;
sum += q*i -i*7;
i ++;
sum += q*i-i*7;
```

Giảm thời gian tính toán

- Trong mạng nơ-ron cổ điển sử dụng hàm kích hoạt sigmoid
- Với x dương lớn hàm bị bão hòa về 1

  `sigmoid(x) ≅ 1`
- Với x âm "lớn" hàm bị bão hòa về 0

`sigmoid (x) ≅ 0`



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

Tính Sigmoid

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

ex = 1+x/1!+ x2/2! + … + xn/n!

- 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 thực dấu phẩy động
- Mạng nơ-ron cổ điển thường gọi hàm sigmoid rất nhiều lần khi thực hiện tính toán.

Tính Sigmoid – Giải pháp

- Xấp xỉ hàm sigmoid bằng spline tuyến tính tại N điểm
  - 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

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})

if ($x < x_0$) return (0.0);

if ($x > x_{99}$) return (1.0);

Tính Sigmoid

- Chọn số các điểm (N = 1000, 10000, …) tùy theo độ chính xác mà bạn muốn
  - Tốn kém thêm không gian bộ nhớ cho mỗi điểm là 2 giá trị float hay double tức là 8 hay 16 bytes
- Khởi tạo giá trị cho mảng khi bắt đầu thực hiện

Kết quả đạt được

- 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 to 640K bộ nhớ.

Lưu ý!

- Với đại đa số 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

- **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.
- **Đơn giản hóa 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 mẩu 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.

Fundamental Rules

- **Code Simplification:** Most fast programs are simple, so keep it simple . Sources of harmful complexity includes: A lack of understanding the task and premature optimization.
- **Problem Simplification:** To increase the efficiency of a program, simplify the problem it solves. Why store all values when you only need a few of them?
- **Relentless suspicion:** Question the necessity of each instruction in a time critical piece of code and each field in a space critical data structure.
- **Early binding:** Move work forward in time. So, do work now just once in hope of avoiding doing it many times over later on. This means storing pre-computed results, initializing variables as soon as you can and generally just moving code from places where it is executed many times to places where it is executed just once, if possible.

Bentley J. Writing efficient programs

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 (tiếp)

- 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. Lập trình viên có thể phải thay đổi điều kiện kết thúc vòng lặp.
    - "Lính gác" hay "Vệ sĩ" 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 1-3 dòng) là inline
  - Tránh phải thực hiện 4 bước khi hàm được gọi
    - Lưu trữ địa chỉ trả về:
      - Địa chỉ lệnh tiếp theo (sau lời gọi hàm) được lưu để phục hồi ngữ cảnh sau khi hàm hoàn thành.
    - Truyền tham số:
      - Các tham số của hàm được truyền vào ngăn xếp hoặc các thanh ghi theo quy ước gọi hàm.
    - Chuyển điều khiển đến hàm:
      - Bộ xử lý nhảy đến địa chỉ của hàm được gọi và bắt đầu thực thi các lệnh trong hàm.
    - Trả về giá trị và phục hồi ngữ cảnh:
      - Sau khi hàm hoàn thành, giá trị trả về được gửi lại cho hàm gọi, và chương trình quay lại địa chỉ lệnh tiếp theo sau lời gọi hàm.
  - Tránh dùng bộ nhớ stack

Optimizing C and C++ Code

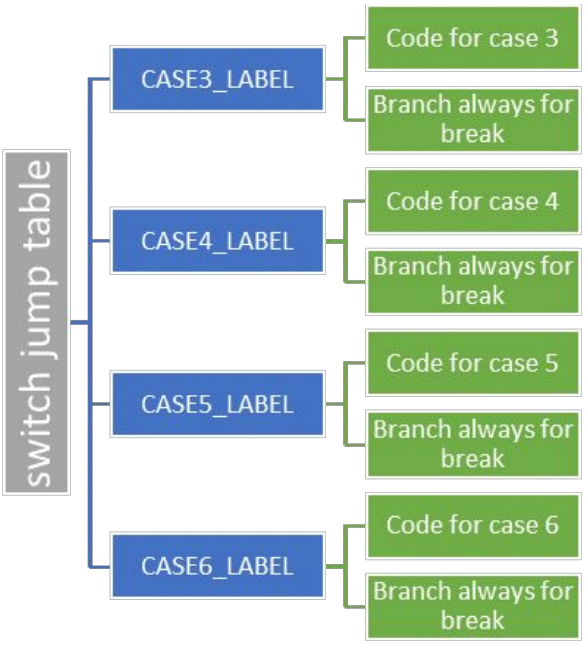
- **Đặt kích thước mảng = bội của 2**

  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 giá trị case của lệnh switch trong phạm vi hẹp**

  Nếu giá trị case trong câu lệnh switch nằm trong phạm vi hẹp, trình dịch sẽ sử dụng jump table để tối ưu hoá lệnh switch.

  - Độ phức tạp gần như O(1)



Optimizing C and C++ Code

- **Đặt các trường hợp thường gặp trong lệnh switch lên đầu**
  - Nếu bố trí các case thường gặp lên trên, việc thực hiện sẽ nhanh hơn
    - Các case đầu tiên sẽ được so trước. Sau khi bắt gặp sẽ thực hiện lệnh trong khối case đó rồi thoát luôn ra khỏi case → Không cần so với các case còn lại.
- **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. Khi đó số phép thử sẽ ít hơn, tốc độ nhanh hơn

Optimizing C and C++ Code (tt)

- **Hạn chế số lượng biến cục bộ**
  - Các biến cục bộ được cấp phát và khởi tạo khi hàm được 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 lớn hơn 8 byte, hãy dùng tham chiếu**
  - Truyền theo kiểu tham trị (pass by value): đối với các kiểu dữ liệu nguyên thuỷ (size <=8 bytes). Vì 1 bản copy của value sẽ được truyền cho hàm → an toàn cho value gốc
  - Truyền theo kiểu tham chiếu (pass by reference): đối với các kiểu dữ liệu lớn (struct, class). Vì chỉ tạo bản copy của tham chiếu (vẫn trỏ đến object gốc) (không copy object) → đỡ tốn bộ nhớ
- **Không định nghĩa giá trị trả về nếu không sử dụng (void)**

Optimizing C and C++ Code (tt)

- **Lưu ý ví trí của tham chiếu tới code và data**
  - Các dữ liệu hoặc code được lưu trong bộ nhớ cache để tham khảo về sau (nếu được). Việc tham khảo từ bộ nhớ cache sẽ nhanh hơn.
  - Vì vậy code và data được sử dụng cùng nhau thì nên được đặt với nhau.
    - Điều này với object trong C++ là đương nhiên. Vì object gồm thuộc tính (data) và phương thức (code)).

Optimizing C and C++ Code (tt)

- **Nên dùng int thay vì char hay short (mất thời gian convert)**
  - Kiểu int có size = 4 Bytes, cùng với kích thước thanh ghi của phần lớn các kiến trúc hiện nay, cũng như kích thước đơn vị truyền (word) giữa cache và bộ nhớ chính
    - → Các lệnh với kiểu < size thanh ghi thường sẽ cần thêm các lệnh bổ sung
  - Quy tắc integer promotion trong C/C++: Khi thực hiện các phép toán với các biến kiểu char hoặc short, CPU sẽ phải chuyển đổi chúng thành kiểu int, và sau khi thực hiện phép toán, có thể sẽ cần chuyển đổi lại thành kiểu ban đầu
- **Nếu biết int không âm, hãy dùng unsigned int**
- **Hãy định nghĩa các hàm khởi tạo đơn giản**
  - Giảm chi phí khởi tạo:
    - Tránh các xử lý không cần thiết, giúp quá trình khởi tạo đối tượng nhanh hơn.
  - Tăng khả năng tối ưu hóa của trình biên dịch:
    - Giúp trình biên dịch dễ dàng inlining và tối ưu mã.
- **Thay vì gán, hãy khởi tạo giá trị cho biến**
  - Nên: (khai báo và khởi tạo luôn)
    - int x = 10; // Khởi tạo trực tiếp
  - Không nên: (thêm bước xử lý (sau khi khai báo có thể x được khởi tạo một giá trị mặc định), sau đó lại phải thêm 1 bước gán với 1 giá trị cụ thể
    - int x; // Khai báo, nhưng không khởi tạo (cấp phát bộ nhớ)
    - x = 10; // Gán giá trị

Optimizing C and C++ Code (tt)

- **Hãy dùng danh sách khởi tạo trong hàm khởi tạo**

```
class MyClass {
public:
    int x;
    MyClass(int val) : x(val) {}  // Khởi tạo trực tiếp thông qua
                                  danh sách khởi tạo
};

#So với cách làm không hiệu quả:
class MyClass {
public:
    int x;
    MyClass(int val) {
        x = val;  // Biến x được khởi tạo trước (với giá trị mặc
                  định) rồi mới được gán lại
    }
};
```

Một vài ví dụ tối ưu mã C, C++

```
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];
```


Một vài ví dụ tối ưu mã C, C++

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

`(unsigned)(x - min) < (max - min)`

Giải thích:

int: $-2^{31} \dots 2^{31} - 1$

unsigned: $0 \dots 2^{32} - 1$

Nếu `x - min >= 0`: Hai biểu thức trên tương đương

Nếu `x - min <= 0`:

`(unsigned) (x - min)` $= 2^{32} + x - min$

$>= 2^{31} >$ `max - min`

Một vài ví dụ tối ưu mã C, C++

```
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 <=

Số thực dấu phẩy động

- So sánh:
  `x = x / 3.0;`
  và
  `x = x * (1.0/3.0) ;`
  (biểu thức hằng `(1.0/3.0)` được thực hiện ngay khi dịch và phép * thực hiện nhanh hơn phép /)

- Hãy dùng `float` thay vì `double`
- Tránh dùng `sin, exp` và `log` (chậm gấp 10 lần *)
- 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

Tài liệu đọc thêm

1. So sánh hàm inline và macro:

https://techdifferences.com/difference-between-inline-and-macro.html

2. Hàm nội tuyến:

https://viblo.asia/p/inline-function-jvElaGRDKkw

3. Tối ưu hóa code C/C++:

https://people.cs.clemson.edu/~dhouse/courses/405/papers/optimize.pdf

4. Jon Louis Bentley Writing efficient programs

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Xin cảm ơn!

soict.hust.edu.vn/ fb.com/groups/soict