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) và cuối cùng **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 và 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 (trình biên dịch) thực hiện **inline expansion (khai triển nội tuyến)** với hàm đó hay nói cách khác là chèn code của hàm đó tại địa chỉ mà nó được gọi.

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 (trình biên dịch) thực hiện **inline expansion (khai triển nội tuyến)** với hàm đó hay nói cách khác là 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;
}
```

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

```
#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)
```

- 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 binh thường.
  - Inline functions có cùng syntax như các hàm khác, chỉ có điều là có thêm từ khóa inline khi khai báo hàm.
  - Các biểu thức truyền như là đối số cho inline functions được tính 1 lần. Biểu thức truyền như tham số cho macros có thể được tính mỗi lần macro được sử dụng.
  - Bạn không thể gỡ rối cho macros, nhưng với inline functions thì có thể.

Biến tĩnh (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 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 bạn gọi lại chương trình con, các biến cục bộ lại được cấp phát và khởi tạo lại.
- Nếu bạn muốn 1 giá trị vẫn được lưu lại cho đến khi kết thúc toàn chương trình, bạn cần khai báo biến cục bộ của chương trình con đó là static và khởi tạo cho nó 1 giá trị.
  - Việc khởi tạo sẽ chỉ thực hiện lần đàu tiên chương trình được gọi và giá trị sau khi biến đổi sẽ được lưu cho các lần gọi sau.
  - Bằng cách này 1 chương trình con có thể "nhớ" một vài mẩu tin sau mỗi lần được gọi.
- Dùng biến Static thay vì Global:
  - Ưu điểm của 1 biến static: biến cục bộ của chương trình con, do đó tránh được các hiệu ứng phụ (side effects).

Stack, 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
    - ví dụ double_array trong ví dụ trên.
- Heap data:
  - Dữ liệu được cấp phát động (ví dụ, pchar trong ví dụ trên).
  - 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)
  do  pos++;
if  (pos >= size) found =0
else found = 1;
```

Dùng "lính canh" ….

- Ý tưởng chung
  - Đặt giá trị cần tìm 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)
 do  pos++;
if  (pos >= size) found = 0
else found = 1;
```

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

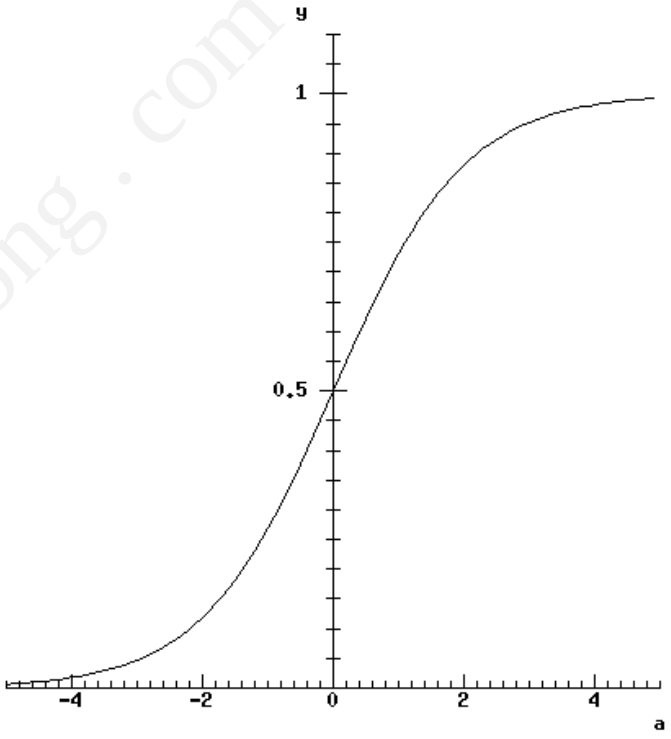
```
for (i =j; i<= j+3;i++)
    sum += q*i -i*7;
```

```
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 sử dụng hàm kích hoạt sigmoid
- Với x dương lớn ta có
  `sigmoid(x) ≅ 1`
- Với x âm "lớn"

  `sigmoid (x) ≅ 0`



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

Tính Sigmoid

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

$$ex = 1+x/1!+ x2/2! + \ldots + xn/n!$$

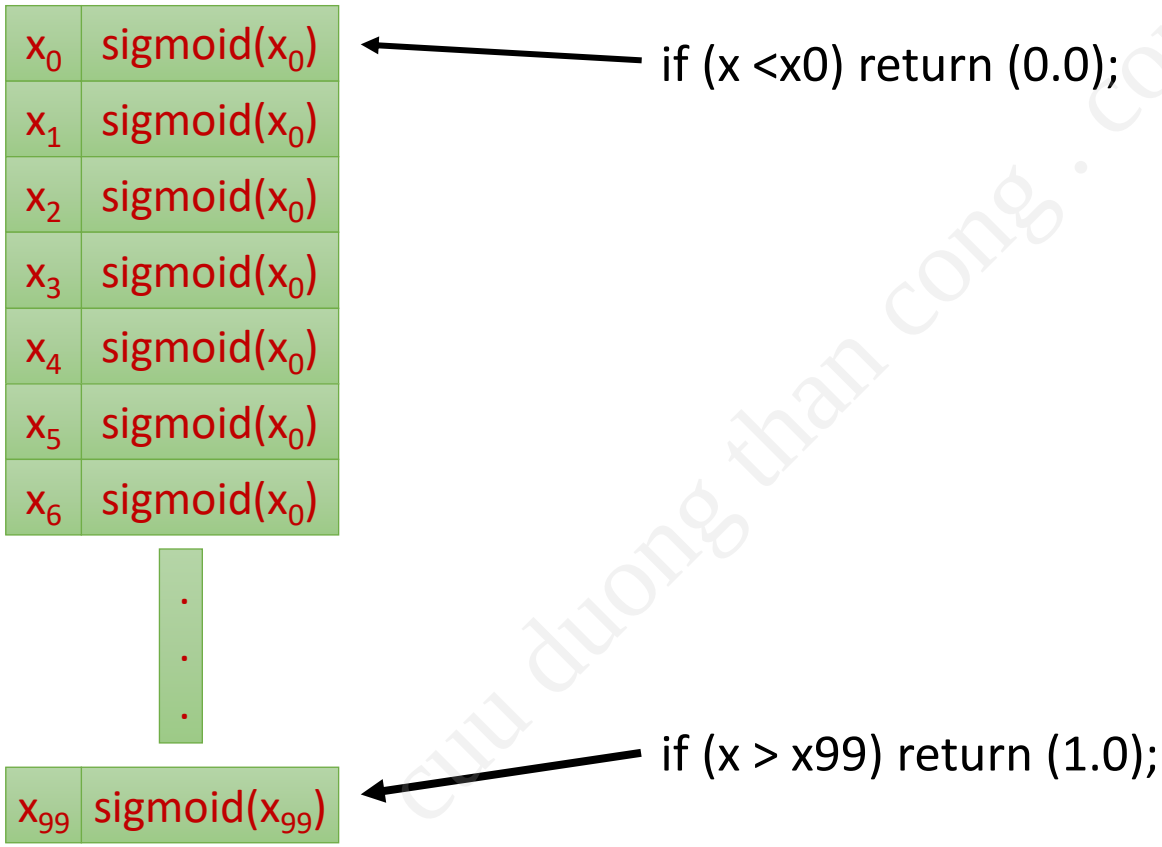
- Hàm exp(-x) mất rất nhiều thời gian để tính!
  - Những hàm kiểu này người ta phải dùng khai triển chuỗi
    - Chuỗi Taylor /Maclaurin
    - Tính tổng các số hạng dạng $((-x)^n / n!)$
    - Mỗi số hạng lại dùng các phép toán với số thực dấu phẩy động
- Mạng nơ-ron thường gọi hàm sigmoid rất nhiều lần khi thực hiện tính toán.

Tính Sigmoid – Giải pháp

- Thay vì tính hàm mọi lúc
  - 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



Tính Sigmoid

- Chọn số các điểm (N = 1000, 10000, v.v.) 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 – 16 bytes
- Khởi tạo giá trị cho mảng khi bắt đầu thực hiện

Tính Sigmoid

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

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 gập 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: 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. 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 dòng) là inline
  - Tránh phải thực hiện 4 bước khi hàm được gọi
  - Tránh dùng bộ nhớ stack

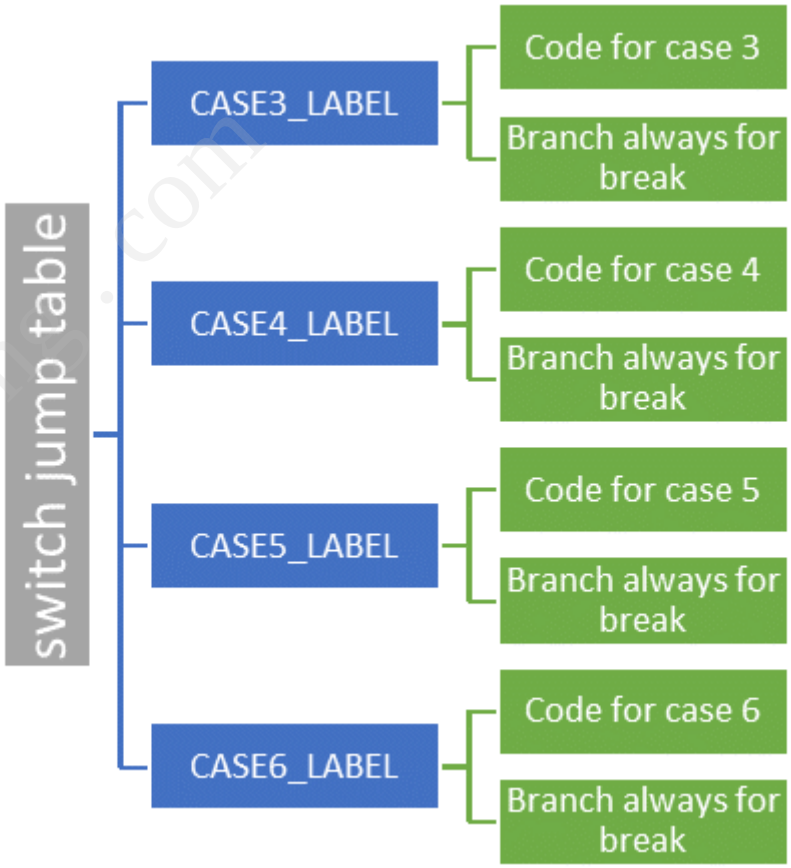
Optimizing C and C++ Code

- **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 jump table. Độ phực 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
- **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 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 lớn hơn 4 byte, hãy dùng tham chiếu**
- **Không định nghĩa giá trị trả về nếu không sử dụng (void)**

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, Stringdesignation){
    m_name = name;
    m_designation = designation;
}
/* === Optimized Version === */
Employee::Employee(String name, String
designation): m_name(name), m_destignation
(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++

```
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: $-2^{31} \ldots 2^{31} - 1$

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

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 được thực hiện ngay khi dịch)

- 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

(Prentice-Hall software series) Jon Louis Bentley - Writing efficient programs-Prentice-Hall (1982).djvu

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