



카테고리



## C/C++ 코드 최적화 팁

C/C++ 2014. 1. 12. 22:50

[뷰어](#) [댓글로](#) [이전글](#) [다음글](#)

### 1 개요

C/C++ 코드 최적화와 관련된 팁들을 적어두기 위한 페이지. 사실 자잘한 테크닉보다는, 제일 느린 부분을 찾아내어 집중적으로 최적화하는 것이 훨씬 중요하지만... 알아서 나쁠 건 없다.

### 2 목록

#### 2.1 구조체의 크기를 2의 승수로 잡아라

구조체의 배열을 인덱싱하는 코드가 있다면, 컴파일러는 구조체의 크기 \* 인덱스와 같은 방식으로 컴파일을 한다. 이 때 구조체의 크기가 2의 승수라면 곱하기 대신에 쉬프트 연산을 사용할 수 있다. 알다시피 곱하기보다는 쉬프트가 빠르다.

#### 2.2 스위치문에 들어가는 케이스의 종류를 줄여라

케이스의 종류가 적은 스위치문이 있을 경우, 요즘 컴파일러는 if-else 구문 대신에 케이스값에 의거한 점프 테이블을 생성한다. 함수 포인터의 배열을 생성한다고 보면 비슷할 것이다. 어쨌든 이 경우, if-else를 여러 개 차례로 체크하는 것이 아니라, 바로 점프를 행하기 때문에 빠르다.

#### 2.3 자주 사용하는 케이스를 스위치문 앞쪽에 뒤라

알다시피 스위치문은 컴파일러에 의해서 if-else 구문으로 변환된다. 코드가 수행될 때는 위에서부터 차례로 체크해 나가면서, 맞아떨어지는 if 구문을 실행하게 된다. 그러므로 자주 사용하는 케이스가 위쪽에 있다면, 밑쪽까지 가지 않아도 되므로 빠르다.

#### 2.4 큰 크기의 스위치문이 있다면, 2중 스위치문으로 나눠라

자주 사용하는 케이스를 스위치문 앞쪽에 두는 것은, 컴파일러에 따라서는 원하는 결과가 나오지 않을 수도 있다. 어떤 컴파일러는 코드 상에 있는 케이스문의 순서를 무시하고 자기 마음대로 순서를 정하기 때문이다. 이런 경우, 2중 스위치문으로 비슷한 결과를 얻을 수 있다.

비교 연산의 횟수를 줄이기 위해서는 자주 사용하는 케이스는 위쪽에다 두고, 자주 사용하지 않는 케이스는 default 케이스 안에다가 스위치문을 하나 더 만들어서 집어넣는다.

스위치문 나누기

Colored By Color Scripter™

```
1 pMsg = ReceiveMessage();
2 switch (pMsg->type)
3 {
4     // 자주 사용하는 케이스는 여기에다 둔다.
5     case FREQUENT_MSG1:
6         handleFrequentMsg1();
7         break;
8
9     case FREQUENT_MSG2:
10        handleFrequentMsg2();
11        break;
12
13    . . .
14
15    case FREQUENT_MSGn:
16        handleFrequentMsgn();
17        break;
18
19    default:
20        // 자주 사용하지 않는 케이스는 이 안에다 둔다.
21        switch (pMsg->type)
22        {
23            case INFREQUENT_MSG1:
24                handleInfrequentMsg1();
25                break;
26
27            case INFREQUENT_MSG2:
28                handleInfrequentMsg2();
29                break;
30
31            . . .
32
33            case INFREQUENT_MSGm:
34                handleInfrequentMsgm();
35                break;
36        }
37 }
```

Thinking Different 구독하기



## 2.5 지역 변수의 숫자를 줄여라

지역 변수의 숫자가 적을수록, 컴파일러가 그 지역 변수들을 레지스터에다가 집어넣을 수 있을 확률이 높아진다. 이는 스택에 있는 변수를 액세스하기 위해서 프레임 포인터에다가 오프셋을 더하는 연산이 없어진다는 말이다. 이는 다음 두 가지의 이유로 인해 상당한 성능의 상승을 가져온다.

지역 변수들이 레지스터에 있으므로, 메인 메모리를 액세스할 필요가 없다. (메모리는 레지스터에 비해서 훨~씬 느리다.) 스택에다 지역 변수를 저장할 필요가 없다면, 스택 생성/삭제, 즉 프레임 포인터 생성/삭제 작업이 없어진다.

## 2.6 지역 변수는 가능한 한 제일 안쪽의 스코프에서 선언하라

지역 변수를 함수 시작 부분 등에다 한꺼번에 선언하지 마라. 즉 코드의 분기에 따라 사용하지도 않을 수 있는 변수들을 한꺼번에 선언해 두지 말고, 필요할 때만 선언해서 사용하라는 말이다.

### 지역 변수의 스코프

Colored By Color Scripter™

```
1 int foo(char* pName)
2 {
3     if (pName == NULL)
4     {
5         A a;
6         ...
7         return ERROR;
8     }
9     ...
10    return SUCCESS;
11 }
```

위의 코드를 보면 알겠지만, pName이 NULL이 아닌 경우에는 A 객체를 생성할 이유가 없다. 그런데 이 A 객체를 함수 시작 부분에다 선언하면, pName이 NULL이건 아니건, 함수를 호출할 때마다 A 객체의 생성자/소멸자가 호출되는 것이다.

## 2.7 파라미터의 숫자를 줄여라

파라미터의 숫자가 매우 많은 경우, 함수 호출할 때마다 많은 양의 스택 푸시가 일어나므로 비효율적이다. 같은 맥락의 이유로 구조체 같은 것을 by value로 전달하지 말아라. 이런 경우, 파라미터 오브젝트 같은 것을 만든 다음, 참조나 포인터로 전달해라.

## 2.8 4바이트(INT의 크기)보다 큰 파라미터나 리턴값을 사용하는 경우에는 참조를 활용해라

파라미터의 숫자를 줄이는 것과 일맥상통하는 이야기인데, 비교적 크기가 큰 파라미터나 리턴값을 by value로 전달할 경우, 스택에서 일어나는 작업이 많아질 뿐만 아니라, 클래스일 경우에는 생성자/소멸자 호출까지 일어난다. 이런 때는 웬만하면 참조를 활용하는 것이 좋다.

참조를 파라미터로 넘길 때, 특별한 경우가 아니라면 const를 붙이는 것이 좋다. 값으로 전달할 때야 함수 내부에서 수정해도 side-effect가 없었겠지만, 참조라면 이야기가 다르기 때문이다.

## 2.9 리턴값을 사용하지 않는다면 아예 void 로 선언해라


쓸데없는 것을 왜 만드는가?

Thinking Different 구독하기



## 2.10 코드와 데이터를 만들 때, locality of reference를 고려해라

C++에서는 클래스라는 개념 때문에, 이것이 어느 정도 자동적으로 보장된다. C에서는 데이터와 데이터를 사용해서 작업을 수행하는 코드가 물리적(같은 소스 코드?)으로 가까운 곳에 위치하도록 하는 것이 좋다.

 <!-- SMP에서는 또 이야기가 다른데... -->

## 2.11 CHAR나 SHORT보다는 INT를 사용하라

INT를 사용해야하는 이유는 C/C++은 기본적으로 산술연산을 int 단위로 처리하기 때문이다. char 범위 안에 들어가는 변수라고 해서 char 타입으로 선언한 뒤에 산술연산을 행하면, 컴파일러가 알아서 int로 변환한 뒤, 연산을 행한 다음, 그 값을 다시 char 타입으로 변환해서 결과를 저장한다. 그러므로 특별한 의미가 있지 않거나, 메모리를 절약해야하는 경우가 아니라면 int를 사용하라.

## 2.12 가벼운 생성자를 선언하라

가능한 한 생성자를 가볍게 만들어라. 생성자는 오브젝트가 생길 때마다 매번 호출된다. 명시적으로 선언한 객체 말고도, 컴파일러가 암묵적으로 생성하는 임시 객체들이 매우 많다는 것을 잊지 않아야 한다. 그러므로 생성자를 가볍게 만들수록, 성능에 있어서 이득을 볼 수 있다.

## 2.13 대입보다는 초기화를 사용하라

다음의 예를 보라.

### Initialization and assignment

Colored By Color Scripter™

```
1 void foo()
2 {
3     Complex c;
4     c = (Complex)5;
5 }
6
7 void foo_optimized()
8 {
9     Complex c = 5;
10 }
```

첫번째 함수에서는 c 객체가 먼저 초기화된 다음, 대입 연산이 일어난다. 두번째 함수에서는 c 객체가 주어진 값으로 바로 초기화가 되어버린다. 즉 기본 생성자 호출이 일어나지 않는 것이다.

## 2.14 생성자 초기화 목록을 이용하라

멤버 변수를 생성자 초기화 목록에서 초기화하면, 위에 있는 예와 마찬가지로 기본 생성자의 호출이라는 오버헤드를 피할 수 있다. 아래의 예를 보라.

Colored By Color Scripter™

```
1 Employee::Employee(String name, String designation)
2 {
3     m_name = name;
4     m_designation = designation;
5 }
6
7 /* == Optimized Version == */
8 Employee::Employee(String name, String designation): m_name(name), m_designation (designation)
9 {
10 }
```

처음 버전에서는 m\_name, m\_designation 변수에 대한 기본 생성자가 일단 호출된 다음, 대입 연산자가 호출된다. 두번째 버전에서는 바로 알맞은 생성자가 호출된다.

## 2.15 "나중을 대비한" 가상 함수를 선언하지 말아라

가상 함수 호출은 일반 함수 호출에 비해 느리다. 그러므로 언젠가 필요할 거라고 생각해서 virtual을 만들 것이 아니라, 필요할 때 virtual을 추가하는 방식으로 코딩해라.

## 2.16 1~3 라인 정도의 함수만 인라인 함수로 선언하라

작은 함수(1~3 라인 정도의 함수)를 인라인으로 바꾸는 것은 성능에 큰 도움을 주는 것이 사실이다. 인라이닝은 함수 호출 대신에 해당 코드를 직접 끼워넣음으로서 성능을 향상시키는 작업이다. 하지만 덩치가 큰 함수를 인라이닝하게 되면, 코드가 커지게 되고, 이는 성능의 저하를 가져온다. (캐쉬 등을 생각하라!) 또한 명심해야할 사항 중에 하나가, 인라이닝을 과용하게 되면 헤더에서 다른 헤더를 참조하는 일이 많아지고, 이는 전체적인 프로젝트를 생각하면 좋지 않다는 것이다.