효율적인 프로그래밍

## 목적

효율적인 C++프로그래밍에 대해 논의한다.

## 목표

해당 프로젝트는 여러 개의 하위 프로젝트로 구분된다. 각 프로젝트는 어떠한 논의거리를 담는다.

## 과정

이 논의들은

1. 더 효율적인 제안  
2. 문제 해결  
의 형태로 등장할 수 있다.

그리고 각 논의에 대하여

1. 달성하고자 하는 어떤 효율적인 목적  
2. 목적에 대한 동기  
3. 그 목적 달성을 위한 제안  
4. 해당 제안에 대한 유효성  
5. 각 제안의 비교와 가장 타당한 제안  
6. 논의 정리와 결론  
7. 실질적인 적용, 구현

위와 같이 7단계로 구분하여 혹은 더 세부적으로 나누어 목적과 그에 대한 결론을 분명히 해야 할 것이다.

[PROJECT1] 효율적인 이동 의미론의 적용

# 달성하고자 하는 목적

효과적으로 자료를 이동시킬 수 있는 방법과 구체적인 사례를 제시한다.

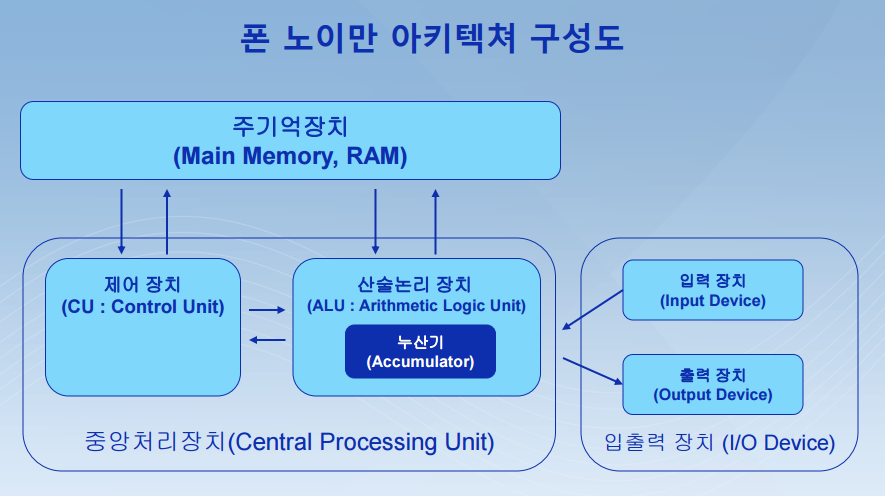
# 목적에 대한 동기

효과적으로 자료를 이동한다는 것은 무엇을 의미하는가?

## 문제에 대한 정확한 인식

아무리 복잡한 프로그램이라도 결국 메모리의 자료를 어떤 순서로 불러오고 저장하느냐에 달려있다.

### 폰 노이만 아키텍쳐



기본적으로 CPU, 메모리, 입출력장치 이 3가지를 결합하여 이루어진 구조

폰 노이만 방식의 **핵심은 “프로그램을 메모리에 내장하는 방식”**이라는 점

기본적 과정은 다음과 같다.

메인 메모리에서 명령어 로딩(fetch) -> 명령어 해석(decode) -> 연산 or 읽기, 쓰기(execute) -> 필요 시, 저장(store)

여기서 어떤 중요한 연산 중 하나가 어떤 자료를 메모리에 담는 과정일 것이다. 이것을 배정이라고 한다.

int a = 5;

a라는 공간에 5라는 자료를 담는다. 이것을 어떻게 효율적으로 담을 것인가가 이번 논의의 주제이다.

## 복사 의미론과 이동 의미론

가장 기본적으로 데이터를 옮기는 방법은 데이터를 복사하여 하나의 임시 객체를 만든 다음, 그 객체를 원하는 공간에 통째로 집어 넣는다. 이것이 복사 의미론이다.

class Archer

{

public:

Archer() {}

Archer(int) {}

Archer(const Archer& rhs)

{

copy\_counter++;

std::cout << "copy\n";

}

Archer& operator=(const Archer& rhs)

{

std::cout << "copy\n";

return \*this;

}

};

class Party

{

public:

Party() {}

Party(const Party& rhs) : box(rhs.box)

{

std::cout << "party copy\n";

}

Party& operator=(const Party& rhs)

{

box = rhs.box;

std::cout << "party copy\n";

return \*this;

}

private:

std::vector<Archer> box{ 1, 2, 3, 4, 5, 6, 7 };

};

int main()

{

Party temp;

auto p = temp;

std::cout << "counter: " << copy\_counter << '\n'; // counter: 14

return 0;

}

위 과정에서 복사를 14번 진행한다. 객체를 생성할 때 기본 값을 만들기 위해 7번의 복사가 일어나며, 다른 객체에 집어넣을 때 또 7번의 복사가 일어난다.

직관적으로 봐도 비효율적이다. 이동만 하고 싶은데 또 하나의 객체를 만든다는 것 자체가 이해할 수 없다. 일단 하나의 객체를 만든다. 이걸 피할 수는 없다. 그러니 한번의 복사는 불가피하다. 임시 객체를 만들지 않고 말 그대로 이동만할 수는 없을까?

더 효율적인 방법은 해당 객체를 저장한 주소를 가져오는 방법이다. 주소만 복사하면 데이터를 모두 복사하지 않아도 그 자료에 접근할 수 있을 것이다.

auto& p = temp;

이제 복사를 7번만 진행할 수 있다.

분명 효율적이지만 한가지 제약이 필요하다. temp가 더 이상 사용되지 않는다는 제약이다. 만약 p가 마음대로 변경되고 있는데 다른 곳에서 temp를 찾는다면 미정의 동작을 맞이하게 된다. p와 temp는 메모리상 같은 위치에 있으므로 정확히 같은 객체인 셈이기 때문이다.

현재 포함된 표현식 다음에 사라지는 임시 객체, 이것이 r-value이다. 이것은 사용자가 메모리에 올린 데이터가 아니다. 컴파일러(?)가 알아서 임시 저장소에 올린 객체이다. 곧 사라질 것이므로 더 이상 사용되지 않음이 보장된다. 그러므로 r-value로 만든 객체라면 포인터만 복사해도 손상의 여지가 없다. 이렇게 r-value의 포인터만 복사하여 옮기는 방법, 이것이 이동 의미론이다.

class Party

{

public:

Party() {}

Party(const Party& rhs) : box(rhs.box)

{

std::cout << "party copy\n";

}

Party& operator=(const Party& rhs)

{

box = rhs.box;

std::cout << "party copy\n";

return \*this;

}

Party(Party&& rhs) : box(std::move(rhs.box))

{

std::cout << "party move\n";

}

Party& operator=(Party&& rhs)

{

box = std::move(rhs.box);

std::cout << "party move\n";

return \*this;

}

private:

std::vector<Archer> box{ 1, 2, 3, 4, 5, 6, 7 };

};

int main()

{

Party temp;

auto p = temp;

std::cout << "copy counter: " << copy\_counter << '\n'; // counter: 14

copy\_counter = 0;

auto p2 = Party();

std::cout << "copy counter: " << copy\_counter << '\n'; // counter: 7

return 0;

}

std::move를 이용하여 box의 포인터만 복사해서 집어넣은 모습이다. 더 이상 모든 원소를 복사하지 않는다. 게다가 더 좋은 것은 따로 메모리를 할당할 필요가 없다는 점이다. 기존에 할당된 메모리를 사용하기 때문이다. 그리고 기존 객체가 손상될 여지도 없다.

# 목적 달성을 위한 제안

위의 정리에 따르면, 가능한 이동하는 것이 무조건 유리하다. 그리고 포인터만 복사하는 이동보다 빠르게 배정할 수는 없을 것이다.

그럼 모든 경우에 이동 연산을 적용하여 복사하지 않고 이동시킬 수 있는가? 이에 대해 몇 가지 조건에 대해 이해해야 한다.

## 옮기는 대상이 r-value이어야 한다.

임시로 생성된, 혹은 앞으로 필요 없을 객체이어야 한다는 것이다. 기존에 있는 객체를 이동할 수는 없다. l-value는 기존의 메모리가 존재하므로 어디선가 사용되고 있다는 의미이다. 그게 아니라면 굳이 메모리에 직접 할당할 이유가 없다.

## r-value라고 하더라도 무조건 이동할 수 없다.

다음과 같은 경우이다.

### 이동 연산이 없다.

이동하고자 하는 객체에 이동 연산이 없는 경우다. 다음과 같다.

class Archer

{

public:

Archer() {}

Archer(const Archer& other)

{ std::cout << "archer copy\n"; }

//Archer(Archer&& other)

//{ std::cout << "archer move\n"; }

};

class Tank

{

public:

Tank() {}

Tank(const Tank& other)

{

std::cout << "tank copy\n";

}

void addArcher(Archer&& one)

{

bunker.push\_back(std::move(one));

}

private:

std::vector<Archer> bunker;

};

int main()

{

Tank t;

t.addArcher(Archer());

return 0;

}

탱크에 아처를 태우는 코드이다. addArcher(Archer&&)를 주목하자. Archer인자를 넘겨서 bunker로의 이동이 일어나야 하지만, 위에서는 복사가 일어난다. 이유는 간단하다. 이동 연산자가 없다. 이동 연산자가 없으면 컴파일러는 오른값을 왼값으로 암묵적 캐스팅하여 복사해버린다.

### 이동이 더 빠르지 않다.

이동은 포인터를 옮기는 과정이다. 물론 포인터가 없는 저장 자료는 없다. 중요한 것은 ‘통째로’ 옮겨야 한다. 한 포인터에 여러 자료가 묶여있지 않다면, 어차피 하나씩 자료를 옮겨야 하므로 기대하는 이동이라는 의미는 맞지 않다.

### 이동을 사용할 수 없다.

이동 연산에 가장 강력한 혜택을 누릴 수 있는 배열. stl자료구조에서는 반드시 이동 연산이 noexcept인 경우에만 이동을 수행하도록 되어있다. 그렇지 않다면 이동 연산이 존재할지라도 복사한다. 이러한 이유는 이동이란 연산은 예외에 매우 취약하기 때문이다. 복사의 경우, 복사 도중에 예외가 발생하면 원본을 되돌릴 수 있지만, 이동은 이미 일부를 옮긴 상태이므로 되돌릴 수 없다.  
그러므로 예외에 안전하지 않다면 이동 연산이 발생하지 않을 수 있다. 참고로 기본값으로 생성된 이동 연산자는 noexcept로 선언된다.

*Question. 분명 noexcept가 아닌데 이동한다?*

분명 noexcept가 아니지만 이동 연산을 수행한다. 하지만 실제로는 noexcept인 것이다. 아마 이동 연산에 예외는 허용될 수 없기 때문에 이렇게 동작하는 것 같다.

# 목적을 달성한 사례 [!다양한 사례]

적절하게 이동이 가능하여 효율을 본 사례를 살펴본다.

## std::vector의 동적 크기 증가

동적 배열은 크기가 저절로 늘어난다. 이는 원소 개수에 비례하여 2배 크기의 새로운 배열을 할당한다. 이 과정에서 모든 원소를 새로운 배열로 복사하는 것보다 이동한다면 훨씬 효율적일 것이다. 기존의 배열은 사라질 것이 보장되므로 기본 조건을 만족한다. 그리고 벡터는 데이터가 들어있는 배열을 포인터로 관리하므로 이동 시 효율성이 보장된다.

## std::emplace로의 삽입

생성자에 필요한 인자를 그대로 전달하면서 임시 객체 생성을 최소화하여 최적화할 수 있다.

# 목적 달성을 위한 구현

위의 제안들을 적용해야 한다. 사용 적절한 시기의 적절한 구현 예시를 제시한다.

template<int n>

class Archer

{

public:

Archer() : pImpl(new impl()) {}

~Archer()

{

if (pImpl) delete pImpl; // 반드시 소멸시켜준다.

}

Archer(const Archer&) { std::cout << "copy\n"; }

Archer(Archer&& other) noexcept // 예외 불가

: pImpl(other.pImpl) // 포인터만 쏙 빼온다.

{

other.pImpl = nullptr; // 이동된 객체의 포인터는 nullptr로 변경해준다.

std::cout << "move\n";

}

Archer& operator=(const Archer&)

{ std::cout << "copy\_op\n"; return \*this; }

Archer& operator=(Archer&& other) noexcept

{

pImpl = other.pImpl;

other.pImpl = nullptr;

std::cout << "move\_op\n";

return \*this;

}

private:

struct impl // 어떠한 자료가 한 곳에 묶여있습니다.

{

int datas[n];

};

impl\* pImpl;

};

int main()

{

std::vector<Archer<100> > group;

for (int i = 0; i < 100; i++)

{

group.push\_back(Archer<100>());

}

return 0;

}

### 더욱 효율적인 이동 구현 [!emplace 구현 예시]

위의 방식은 Archer객체를 만들면서 생성자를 호출한다. 그리고 이 객체를 벡터로 이동하는데, 그 벡터에서는 Archer을 또 생성해야 한다. 그리고 이 생성된 객체에 대상의 포인터만 취하게 된다.

여기서 생성자를 2번 호출한다는 점을 알 수 있다. 꼭 이래야 할까? 매개변수만 이동시켜서 1번의 생성자 호출으로 객체를 이동시킨다. 이것을 가능하게 해주는 재료로 가변 인자 템플릿, 완벽 전달, 이동 연산이 존재하는 것이다.

int main()

{

std::vector<Archer<100> > group;

for (int i = 0; i < 5; i++)

{

group.emplace\_back(/\*인자들\*/);

}

return 0;

}

옮겨질 장소가 stl이 아니더라도, 완벽 전달로 인자를 넘겨서 생성 호출을 최소화하는 전략은 중요하다.

# 논의 정리 및 결론

가능하면 이동하되, 할 수 없다면 복사한다.

이동으로 인한 최적화의 핵심은 결국 한번 할당한 자료를 최대한 활용한다는 것이다. 이는 곧 재활용이며, 효율적이게 된다.

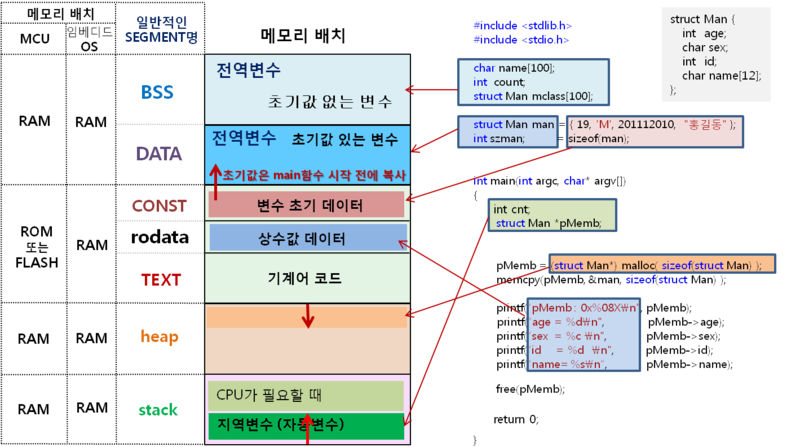
[PROJECT2] 기본 제공 타입: int와 double, long long의 비교

# 달성하고자 하는 목적

각 타입의 연산속도차이는 어떨까? 혹은 각 타입의 이점은 무엇인가?

# 이론적 배경

## 메모리 구조



# 결론

참고: http://hashcode.co.kr/questions/2048/c-실수-연산-정수연산-속도-차이

-1. 정확한 계산을 생각하신다면 정수형으로 해야합니다.

-2. 실수형은 컴퓨터의 특성상 이진으로 처리하기 때문에 연산을 거듭할 수록 오차가 누적됩니다.

#include <stdio.h>

int main(){

float a = 0.0001,b=0;

int i;

for(i=0;i<1000;i++) {

b = b + a;

}

printf("b = %e",b);

printf("b = %f",b);

return 0;

}

양식의 맨 위



양식의 맨 아래

위 예를 보면, float의 경우 유효자리를 표현하기 위한 bit수가 매우 적어 더 극명하게 잘 보여줍니다. 0.0001을 1000번 더하면 0.1이 되어야 하지만, 위를 돌려보면 0.1이 되지 않는 걸 알 수 있습니다.

-3. 의미적으로 실수를 사용해야만 하고, 정확도를 요구한다면 double 형을 써야합니다. 이 경우에도 어떤 계산의 결과를 기대할 때, 유효자리를 산정하여 맞는지 검사해야 합니다. 예를 들어 어떤 실수 연산 결과가 소수점 이하 5자리까지만 맞는지 검사한다는 식의 코드를 작성해야 합니다.

-4. 위 2,3과 같은 이유로, 실수는 정확한 카운트를 해야 하는 (ex. for loop의 count 변수) 경우에는 사용하지 말아야 합니다.

-5. 속도는 기본적으로 우리가 사용하는 컴퓨터의 하드웨어에서 같은 장치 안에 포함된 기능을 사용한다면, 정수형 계산이 실수형 계산보다 빠릅니다. 실제로는 부동소수(실수) 계산을 위한 하드웨어(예: GPU)들은 때때로 더 빠른 성능을 내기도 함으로, 어떤 하드웨어를 사용하느냐 달려있다고 볼 수 있습니다. 비단 이 뿐만 아니라 데이터 사이즈(16bit, 32bit, 64bit ...) 에 따라서, 메모리에 접근횟수가 달라짐으로, 우리가 사용하는 알고리즘(혹은 프로그램)에서는 생각보다 큰 차이를 보이지 않을 수도 있기 때문에, 상황에 따라 달라질 것 같습니다.

[PROJECT3] static키워드의 사용

# 달성하고자 하는 목적

static혹은 전역 변수를 사용해도 성능에 아무런 문제가 없는가?

# 이론적 배경

사람1:

1. 지역변수는 초기화 여부를 검사하기 위해 분기가 일어나므로 **전역변수보다 느립니다.**  
2. static 변수는 캐시상에 없을 확률이 매우 높습니다.  
3. volatile 이나 static 키워드가 붙게 되면 대부분의 컴파일러 최적화 대상에서 제외됩니다. (그래서 퍼포먼스 테스트시에 불필요한 최적화를 막기 위해 volatile 이나 static 키워드를 붙여주기도 합니다.)  
4. 여러 쓰레드에서 코드를 접근 할 때 쓸 때 참사가 발생합니다.

사람2:

일단 VC++ 을 기준으로 본다면(gcc도 별로 다르지 않으리라 생각됩니다.) 함수가 한번 호출되면 inline 되지 않는 이상 항상 stack 의 크기가 커지게 됩니다. 그런데 이때 변수의 개수에 상관 없이 컴파일 시점에 결정된 변수의 총 크기만큼 stack이 늘어난다는 것입니다. 인자가 1개이건 100개이건 stack이 늘어나는데 필요한 비용은 동일하다고 볼 수 있습니다. 여기서 page fault 가 날 가능성이 아주 약간 있지만 OS 에서 stack 을 할당할때는 미리 commit 된 상태로 준비해주기 때문에 일반적이지 않습니다.   
  
일단 stack의 크기.. 즉 지역변수의 개수에 따른 패널티가 없다고 치면 접근에 대해서 살펴볼 차례입니다. 만약 변수가 지역변수, 즉 stack에 들어가는 변수라면 컴파일러는 아주 아주 공격적인 최적화를 수행합니다. 가령 배열이나 구조체를 보면 쉽게 알 수 있는데 일반적인 포인터나 전역의 경우 구조체나 클래스의 첫번째 포인터를 찾은 다음 그것에서 n 바이트만큼 더한 정보를 읽어오게 됩니다. 그렇지만 stack의 구조체의 경우 구조체의 첫 부분도 찾지 않고 곧바로 stack의 시작에서 n 바이트 떨어진 곳의 데이터를 읽어오게 됩니다.   
  
별것 아니라고 생각할 수 있지만 구조체의 위치에 의존해서 병렬화 될 수 없는 두개의 명령을 투명하게 실행 가능한 한개의 명령어로 대체한다는건 큰 이득이라고 할 수 있습니다. 저는 이런 문제로 루프에서 아주 자주 참조되는 클래스의 멤버 변수가 있다면 일단 지역 변수(stack)으로 복사본을 만든다음 접근하도록 하고 있습니다. 리스팅 파일을 참조하도 차이가 있고 명령어들이 더욱 효율적으로 병렬화 될 수 있도록 해주는데 있어서 절대 망설일 필요가 없는 작업이기 때문이죠.  
  
일단.. 물리적인 메모리에는 속도 차이가 없지만 컴파일러가 생성한 코드에 따라서 접근성이 가벼워지는 순서는 지역변수-전역변수-포인터 순서라고 생각하면 됩니다. 일반적으로 '어느 메모리가 빠르냐'라는 질문의 답변도 위와 같습니다.

# 연구 과정

## 첫 번째 비교

단순한 산술 연산을 반복했을 시에 지역 변수와 어떤 차이가 있는지 시간을 측정한다.

static int var1 = 0;

static const int count = 100000000;

int func1()

{

var1 = 5;

for(int i = 0; i < count; i++)

var1 \*= 98 + 5 - 11 % 3;

return var1;

}

int func2()

{

int var2 = 5;

for (int i = 0; i < count; i++)

var2 \*= 98 + 5 - 11 % 3;

return var2;

}

int func3()

{

int\* var3 = new int(5);

for (int i = 0; i < count; i++)

\*var3 \*= 98 + 5 - 11 % 3;

return \*var3;

}

void measureTime(int idx, int (\*func)())

{

std::chrono::system\_clock::time\_point start = std::chrono::system\_clock::now();

func();

std::chrono::duration<double> sec = std::chrono::system\_clock::now() - start;

std::cout << idx << "번째 함수 걸린 시간(초) : " << sec.count() << " seconds" << std::endl;

}

int main()

{

measureTime(1, func1);

measureTime(2, func2);

measureTime(3, func3);

return 0;

}

걸론: 아무런 차이가 없다.

## 두 번째 비교

지역성이란 것은 메모리에서 비슷한 위치에 있는 값들을 참조하는 것을 말합니다. 메모리 주소가 0부터 127까지 있다고 할 때, 메모리 액세스 패턴이 10, 11, 9, 10, 10, 12, 10, ... 이런 식으로 되면 지역성이 높은 것이고, 0, 115, 40, 66, 77, 1, 101, ... 이런 식으로 되면 지역성이 낮은 것입니다. 캐시에 다 들어갈 수 없는 데이터를 액세스하는데 지역성이 낮으면 캐시 미스 레이트가 높아지겠죠.

그리고 역시 바이너리를 보기 전에는 확언할 수 없지만 모든 변수가 스택에 있을 경우가 캐시 적중률이 약간 더 높을 것으로 보입니다. 메인 함수의 루프 안에서 scanf와 printf를 지속적으로 호출하는데 함수 호출은 스택 영역에서 값을 쓰는 것으로 이루어집니다. 호출될 함수의 주소나 넘겨질 인자의 값 등이 스택에 쓰여지죠.

static const int count = 100000;

int arr\_static[count];

int arr2\_static[count];

int i\_static = 0, j\_static = 0;

int func1()

{

int arr[count];

int arr2[count];

int j = 0;

for (i\_static = 0; i\_static < count; i\_static++)

{

if (i\_static > 50) i\_static++;

int k = i\_static + j;

arr\_static[i\_static] = arr2[j];

j += i\_static;

j %= count;

if (i\_static > 100) i\_static--;

std::cout << i\_static;

}

std::sort(arr\_static, arr\_static + count);

return arr\_static[50];

}

int func2()

{

int arr[count];

int arr2[count];

for (int i = 0, j = 0; i < count; i++)

{

if (i > 50) i++;

int k = i + j;

arr[i] = arr2[j];

j += i;

j %= count;

if (i > 100) i--;

std::cout << i;

}

std::sort(arr, arr + count);

return arr[50];

}

결과: 전역12초, 지역 3초

# 결론

한 스택에서 꽤나 복잡한 함수를 호출한다면 이는 역시 스택에서 일어나는데 지역성에 영향을 미친다. 호출한 스택에서 자기 일을 하느라 이전에 저장한 캐시를 죄다 밀어버리기 때문이다.