

소프트웨어 개발자 이야기 - C언어에 대해



2017년 12월 20일 수요일 - Version 1.0.0

저자 : 권수호(suho.kwon@gmail.com)

Copyright @2015, 2016, 2017...(계속)

본 문서의 내용을 인용하거나 상업적인 목적으로 사용하기 위해서는 저자와의 협의가 있어야 합니다.

- 감사합니다.-

머리말

소프트웨어 개발자로 어느새 20년을 살아왔다. 첫 번째 책을 쓰고 난 이후로 15년이란 세월이 흘렀고, 이제는 그 동안 알게 되었거나 느꼈던 것을 글로 남겨야 할 때가 왔다고 생각한다. 누구를 위해서 이 글을 쓴다고 이야기 하는 것은 내 자신에 대한 지나친 칭찬이 될지도 모른다. 그냥 이 글은 나 자신을 위해서 쓴다고 하는 편이 더 솔직할 것이다. 그 동안 했던 많은 실수들과 실패의 기록이며, 왜 그렇게 하지 못했을까하는 후회의 유산일 뿐이다. 똑 같은 실수와 후회를 반복할지도 모를 사람들을 위해서는 작은 도움이 될 수 있을지도 모르지만, 이미 잘 알고 있는 전문가들을 위해서는 아까운 시간과 지면의 낭비가 될지도 모르겠다.

하지만, 그래도 글을 써야 한다고 생각한 것은 그 동안의 경험을 통해서 “우리는 제대로 프로그램을 개발하고 있지 않다”라는 것을 느꼈기 때문이다. 특히, 제대로 된 경험을 하지 못한 상태에서 성급하게 뛰어드는 상업적인 소프트웨어 개발은, 불안정한 코드를 만들어 지속적으로 개인의 삶에 다양한 영향을 주기 때문이다. 문제를 사전에 찾아내고 수정해야 하지만, 대부분의 개발자는 문제를 어떻게 예방해야 하는지 조차 모른다. 이런 저런 웹사이트나 초급의 책에서나 볼만한 예제 코드를 실무에 아무런 비판없이 사용한다. 소위 말해서 하지 말아야 하는 많은 코드들이 상용화 된 코드에서 발견되는 이유가 바로 여기에 있다. 따라서, 그런 코드들이 발생하는 원인과 그것을 해결할 수 있는 방법을 오래 동안 고민할 수 밖에 없었다.

이미 이런 이야기를 하는 책은 세상에 많이 있지만, 실무와 동떨어져 있거나(다른 언어로 작성되어 있거나), 혹은 실용적인 가르침을 주기에 부족한 부분이 많이 있는 것으로 생각되었다. 따라서, 이 책을 통해서 이루고자 하는 것은 코드를 어떻게 하면 더 “아름답게” 만드는 방법을 알려주는 것이 아니라, 코드를 통해서 실질적인 비용과 낭비를 줄이는 것이다. 새로운 수익을 만드는 것은 창의적인 생각과 사용자가 원하는 것을 해결하는 것이며, 비용과 낭비를 줄이는 것은 “해서는 안되는 일을 하지 않는 것”이다. 소프트웨어 개발에서 발생하는 낭비의 대부분은 “재작업(Rework)”이며, 그것을 최대한 억제할 수 있는 방법으로 코드를 만드는 것이 결국 더 많은 이윤을 남긴다는 것이다.

문제는 간단하다. 버그를 만들 가능성이 낮은 방법으로 코딩하는 것과 만들어진 코드를 수시로 수정하고 점검하는 것이다. 하지만, 간단한 것들을 간단하게 이루지 못하는 것이 세상의 이치다. 간단한 것들은 사람의 끊임없는 주의를 요구하며, 잠시 게을러지면 그 틈을 파고들고 잘못된 습관으로 고착되기 때문이다. 새로운 개발자가 잘못된 습관이 고착되면 20년 후에도 이런 글을 읽고 느끼지 못하는 사람이 될 것이다.

적어도 조금이라도 자신의 삶이 의미있었다는 이유를 만들고자 한다면, 잘못 알고 있었던 지식들을 검증할 수 있도록 공개하는 것이 옳다. 간단한 문제는 의외로 간단하게 해결될 수도 있다. 다만 해결 방법에 차이가 나는 것은 “생각의 변화”를 필요하기 때문이다. 기존의 문제를 기존의 방식으로 풀려고 애쓴다면, 결국 기존의 실패를 반복할 뿐이다. 물론, 그 실패를 인정하지 않을지도 모르지만, “반복된 실패는 실력이다”. 이제는 문제를 보는 시각을 넓혀 문제의 근본을 해결하려고 나서야 할 때인 것이다.

이 책은 초급자를 위한 것이 아니다. 이미 C언어를 이용해서 개발을 하고 있는 사람들을 대상으로 한다. 하지만, 그렇다고 초급자가 볼 수 없다는 것도 아니다. 중급 혹은 고급 개발자로 발돋움 하고 싶은 초급 개발자라면 반복해서 읽어도 무방하다. 물론, 한 번 읽고나면 더 높은 수준의 책이 필요하겠지만, 이번에는 이 정도로 만족 하기를 빈다. “더 좋은 것들 항상 미래에 있다”고 이미 누군가가 이야기를 했다. 따라서, 이 책에 적혀있는 생각들이 일반화 될 정도로 성숙하게 된다면, 더 좋은 방법이 세상의 어딘가에는 반드시 존재하고 있을 것이다. 우리의 이번 여행은 그것을 찾아내기 위한 작은 시작일 뿐이다.

- 일요일 오후, 아이스 카페라테와 함께한 시간의 흔적을 남기다.-

머리말	3
1. 프로그래밍 이란?	11
[프로그램을 처음 작성하는 사람들에게]	12
[작은 것을 잘 만들어야 한다.]	17
[코드의 품질이란?]	20
[프로그래머]	22
[과제에서 살아남는 방법]	23
[테스트의 중요성]	25
[“좋은 코드”가 가지는 특징]	27
[개발 툴(Tool)에 대해서]	28
[개발환경 익히기]	31
[“Hello, World!!!”를 다시 보다.]	36
[“Hello, World!!!”를 더 자세히 보기]	39
[생각의 추상화와 계층화]	41
[절차 지향과 객체 지향]	44
[C언어 표준에 대해서]	45
[C99표준의 printf() 함수 적용]	47
[이름 없는(Anonymous) 구조체의 사용]	48
[버전 관리 시스템의 사용]	49
[“Doxygen”을 이용한 코드의 문서화]	50
[UML을 이용한 코드 실행의 분석]	53
[좋은 코드의 특성]	54
[관리하기 쉬운 코드를 만드는 원리]	57
[소프트웨어 아키텍처(Software Architecture)]	60
[플랫폼(Platform) vs. 프레임워크(Framework)]	60
[좋은 이름 만들기]	61
[함수(Function), 프로그래밍의 최소단위]	63
[연산자의 우선 순위를 걱정하나?]	65
[포인터(Pointer)에 대해서]	65
[재귀(Recursive) 호출]	66
[전역변수의 사용]	69
[문법보다는 눈에 잘보이는 코드 작성하기]	70
2. 코딩 룰	73
[명료한 코드 만들기]	74

[코딩 룰(Coding Rule) 자세히 보기]	75
[소스 코드의 관리]	89
[컴파일러의 경고는 전부 제거해야 한다.]	90
[구조화를 위한 코드 작성]	91
[변수와 연산자의 적절한 타입]	92
[변수의 사용 범위를 좁게 만들기]	93
[지나치게 큰 지역 변수를 사용하지 않기]	94
[방어적인 코딩]	94
[함수의 인수 개수를 최소로 유지하기]	95
[메모리 할당과 해제]	96
[위생검사(Sanity Check)하기]	98
[매크로를 활용한 디버깅(Debugging) 메시지의 출력]	98
[모듈별 디버깅 매크로의 정의]	99
[안전한 코딩(Secure Coding)에 대해]	100
[스트링(String) 관련 함수를 사용할 때 주의할 점]	101
[경계조건에 대한 주의]	101
[3차원 이상의 배열 사용하지 않기]	102
[포인터 연산도 연산자의 우선 순위를 보이게 만들어라.]	104
[“for()” 반복문의 사용법]	105
[열거형(Enumerator)을 이용한 상수값(Magic Number)의 대체]	106
[이름 만들기]	108
[모든 변수는 정의 할 때 값을 정하라.]	110
3. 코드 리뷰	112
[읽기 쉬운 코드가 좋은 코드다.]	113
[책임(혹은, 계약)과 구현의 분리]	115
[코드 리뷰를 못하는 이유?]	116
[코드 리뷰 하는 방법]	118
[코드 리뷰 체크 리스트(Checklist)]	120
[변수(Variable)에 대해]	122
[변수의 타입에 대해서]	124
[암시적인 타입 변환 보다는 명시적인 타입 변환을 사용]	125
[타입 캐스팅(Type Casting)을 주의]	127
[함수(Function)에 대해]	128
[함수의 출구는 몇개가 좋을까?]	130

[파일(File)과 디렉토리(Directory)에 대해]	131
[코드의 복잡성(Complexity) 문제를 해결하는 방법]	132
[코드 리뷰 가이드 Version 1.0]	133
[코드 리뷰 가이드 Version 2.0]	137
4. 프로그램의 구조	140
[모듈화란?]	140
[캡슐화(Encapsulation)란?]	141
[계층화(Layering)란?]	142
[계층구조에 대한 단상]	143
[데이터의 상태 변경에 대한 주의]	144
[디렉토리(Directory)]	145
[파일]	146
[코드]	147
[주석 사용법]	149
[스타일]	149
[헤더 파일의 위치]	151
[조건문 없애기]	152
[자료구조 감추기]	153
[전역 변수]	155
[조건식(Conditional Expression)]	156
[추상화(Abstraction)란?]	158
[유지보수(Maintainability)를 쉽게 만들어주는 코딩]	159
[파일의 구성]	161
[파일간의 의존성 낮추기]	163
[한 줄에 하나의 명령어만 사용]	164
[조건문에 할당문이 들어가지 않도록 주의]	165
[조건문에서 불리언(Boolean)값만 사용한다.]	165
["switch()"을 제대로 사용하기]	167
["continue", "goto", "break"의 사용 줄이기]	168
["float"의 사용에 대한 주의]	170
[변수의 크기(Size)에 민감한 코드]	170
[할당문과 비교문]	172
["const"의 사용]	172
["static"과 "const"의 사용]	175

[매크로(Macro)는 간단한(?) 것만]	176
[매크로(Macro)의 활용]	178
[매크로 사용 줄이기]	180
[“#define”의 대체 “enum”]	181
[인라인(Inline) 함수의 사용]	181
[“assert()”의 활용]	183
[함수의 호출값 확인]	183
[조건문은 적게 사용하라]	184
[조건문의 분기를 간단하게 만들기]	187
[조건문을 평평하게(Flat) 만들기]	188
[“switch()”의 사용 줄이기]	190
[해쉬(Hash)를 이용한 “switch()”문의 대체]	190
[“#pragma once”의 사용]	194
[추상화의 적용]	195
[구조체와 함수의 결합]	196
[콜백(Callback) 함수의 구현]	198
[배열의 사용]	199
[배열을 함수의 인자로 사용하기]	200
[포인터로 넘기고, 배열로 쓰기]	201
[배열의 크기를 넘길 필요가 없도록 만들기]	202
[배열은 간단하게 사용]	203
[배열의 인덱스(Index)는 자주 문제를 일으킨다.]	205
[배열 사용시 오버플로우(Overflow)의 주의]	206
[구현은 숨기고 선언은 보여주기]	207
[인터페이스를 이용한 구현의 분리]	209
[헤더 파일의 사용]	211
[C구현 파일 나누기]	212
[헤더(Header) 파일의 위치와 내용]	215
[조건부 컴파일 줄이기]	217
[조건부 컴파일의 제거]	218
[코드에 계층구조 도입하기]	219
[하나의 파일에는 하나의 외부 공개 함수만 두기]	221
[함수의 길이를 짧게 만드는 방법]	222
[함수의 명확성을 높이는 역할 분리]	224

[“goto”문의 올바른 사용]	227
[참조(Reference) 및 호출(Call)]	228
[역할과 책임(Role & Responsibility)의 구현]	231
[자료구조 숨기기]	233
[NULL Object를 활용한 검사 코드의 제거]	236
[리플렉션(Reflection)을 이용한 자신의 자료구조 접근]	240
[스택(Stack)의 활용]	243
[큐(Queue)의 활용]	245
[포인터를 이용한 스택의 구현]	248
[포인터를 이용한 큐의 구현]	250
[포인터를 이용한 큐의 구현 개선]	253
[커멘드 라인(Command Line)의 해석]	256
[직접 하려고 하지 말고 시켜라(역할과 책임의 분리)]	259
5. 단위 테스트	261
[단위 테스트 프레임 워크]	261
[하드웨어가 없으면 코딩할 수 없다?]	264
[테스트를 위한 코드 작성 법]	265
[의존성의 주입(Dependency Injection)]	266
[의존성의 주입의 활용]	268
[테스트를 위한 코드 작성 규칙]	270
[Unity를 이용한 C 단위 테스트]	277
[CMock을 이용한 단위 테스트]	285
[CppUTest를 이용한 단위 테스트]	295
[CppUMock을 이용한 단위 테스트]	299
[테스트 범위(Test Coverage)의 측정]	305
[Unity와 CppUTest의 비교]	307
[GTest를 이용한 단위 테스트]	308
[GMock의 활용]	319
6. 디자인 패턴	331
[간단한 UML(Unified Modeling Language) 익히기]	332
[디자인 패턴의 분류 방법]	335
[상태 패턴(State Pattern)을 이용한 이벤트(Event)의 처리]	335
[어댑터 패턴(Adapter)을 이용한 차이의 극복]	346
[추상 팩토리 패턴(Abstract Factory Pattern)을 이용한 다양한 제품 만들기]	350

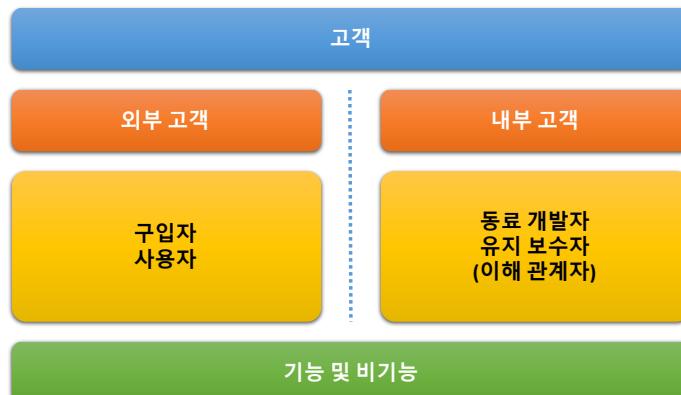
[장식자 패턴(Decorator Pattern)을 이용한 꾸미기]	354
[책임 연쇄 패턴(Chain of Responsibility)를 이용한 공유된 자원 이벤트의 처리]	356
[파사드 패턴(Facade Pattern)을 이용한 내부 구현 감추기]	358
[컴포짓 패턴(Composite Pattern)을 이용한 자료구조의 둑음 다루기]	362
[전략 패턴(Strategy Pattern)을 활용한 알고리즘 변경]	366
[단일자 패턴(Single Pattern)을 이용한 공유 자원 다루기]	369
[대리자 패턴(Proxy Pattern)을 이용한 권한의 위임]	372
[템플릿 메소드 패턴(Template Method Pattern)을 활용한 다양성의 구현]	375
[방문자 패턴(Visitor Pattern)을 이용한 연결된 자료구조의 원소 다루기]	377
[브릿지 패턴(Bridge Pattern)을 이용한 모듈간의 연결고리 만들기]	382
[플라이 웨이트 패턴(Fly Weight Pattern)을 이용한 작은 자료구조들의 생성]	386
[반복자 패턴(Iterator Pattern)을 이용한 개별 원소 다루기]	390
[관찰자 패턴(Observer Pattern)을 이용한 변경 통보받기]	394
[중재자 패턴(Mediator Pattern)을 이용한 통신의 구현]	398
[프로토타입 패턴(Prototype Pattern)을 이용한 복제(Clone) 생성]	403
[커멘드 패턴(Command Pattern)을 이용한 명령어 처리]	405
[팩토리 메소드 패턴(Factory Method Pattern)을 활용한 다양한 제품의 생산]	409
[빌더 패턴(Builder Pattern)을 이용한 개별 자료구조의 생성]	413
[통역자 패턴(Interpreter Pattern)을 이용한 문장의 해석]	416
[메멘토 패턴(Memento Pattern)을 이용한 “Undo”의 구현]	421
7. 최적화	427
[최적화 전에 최적화가 필요한지 확인하라.]	429
[최적화를 위한 접근 방법]	429
[개별적으로 동작 하면 문제가 없지만 동시에 동작 하면 성능 문제가 생길 수 있다.]	433
[전역 변수 사용 줄이기]	433
[모든 변수는 기본적으로 정수로 선언하는 것이 좋다.]	434
[코드의 실행 시간 구하기]	435
[정밀한 실행 시간 얻기]	436
[함수의 호출 경로(Call Path)가 길어지지 않게 만들기]	439
[메모리 복사 줄이기]	442
[전역 변수와 지역변수의 비교]	443
[“volatile” 변수와 최적화]	445
[루프 풀어(Loop Unrolling) 해치기]	447
[배열 초기화에 대한 최적화]	448

[배열 접근에 대한 최적화]	450
[“switch()”의 최적화]	452
[반복문 내에서 함수 사용하지 않기]	454
[반복문 내에서 조건문 줄이기]	454
맺음말	457

1. 프로그래밍 이란?

프로그램을 만드는 것을 프로그래밍이라고 한다. 프로그램은 컴퓨터가 이해하는 언어로 코드를 만들어진 0과 1의 연속이다. 하지만, 이렇게 정의하고나면 정작 중요한 요소가 빠지고 만다. 즉, 프로그래밍은 사람이 하는 것이며, 사람이 만든 컴파일러를 이용해서 컴퓨터가 이해하는 “코드”로 변환된다는 것이다. 그리고, 그것을 사용하는 것도 사람이 주요 대상이며, 유지보수를 담당하고 있는 것도 사람이다. 전부 사람의 일이라는 것이다. 가장 중요한 것은 사용자의 요구를 해결하는 것이 프로그래밍의 주된 역할이라는 점이다. 프로그래밍이란 사용자가 필요한 가치를 프로그램으로 구현해서, 사용자에게 즐거움을 제공하는 활동이다.

시중에는 C언어에 대한 다양한 종류의 책들이 많이 나와 있다. 그리고, 많은 프로그래머들이 그런 책을 읽고 공부한 후에 프로그래밍을 시작 한다. 다양한 책들의 예제는 신참 프로그래머에게는 따라하기 쉬운 코드들로 이루어져 있고, 아무런 부담감 없이 "Copy-and-Paste"를 한다. 이런 습관이 자주 들다보면 결국 상업적으로 코딩해야 할 경우에도 책에서 나온 예제를 그냥 사용하게 되고, 부지불식간에 익힌 습성이 몸에 밴다. 하지만, 상업적으로 유용한 코드를 짜는 것은 다른 세상이며, 그런 코딩을 가르치는 곳도 없다. 오로지 자신의 "시행착오"와 오랜 경험 및 공부를 통해서만 배울 수 있는 지식은 쉽게 얻지 못하는 값비싼 것들이다. 이제는 이런 부분에 대해서 충분히 이야기 할 필요가 있다고 본다. 왜냐하면, 우린 일상에서의 실패가 모두 이런 것들에 기인하고 있다는 사실을 잘 알기 때문이다.



상업적인 코드와 "아마추어적인 코드"의 차이점은 뭘까? 단순히 생각한다면, 상업적인 코드는 외부 고객(External Customer)뿐 아니라, "내부 고객(Internal Customer)"의 욕구를 만족 시켜줄 수 있는 코드이며, 내부 고객으로는 다른 소프트웨어 개발자와 상품기획, 관리자, 테스터등 다양하다. 다른 소프트웨어 개발자들은 코드가 읽기 편하고 잘 구조화(Structured)되어 있어서 전체적으로 이해하기 쉬워야 한다는 것을 강조한다. 상품기획은 추가적인 기능 구현을 꾸준히 요구하기에, 확장이 쉬운 코드를 만들기를 기대하며, 관리자는 저비용으로 짧은 기간동안에 원하는 소프트웨어를 만들어내기를 소프트웨어 개발자에게 지시 한다. 테스터도 마찬가지로 자신이 테스트 해야할 코드가 테스트하기 쉽게되어 있고, 자동화된 테스트가 있는 것을 원할 것이다.

가장 중요한 것은 어쨌든 나 이외의 다른 소프트웨어 개발자가 편한 마음으로 쉽게 고칠 수 있는 코드를 만들어야 한다는 점이다. 따라서, 이런 것들이 궁극적인 상업적으로 좋은 코드라 할 수 있겠다. 물론, 상업적으로 만들어진 코드들이 모두 이런 것들을 만족 시킨다고 볼 수는 없다. 하지만, 추구해야할 목표는 반드시 "이해하기 쉬운"코드 이어야 한다.

이해하기 쉬운 코드를 만드는 것은 기초적인 문법 만을 가지고 되는 것은 아니다. 즉, 많은 시행착오를 겪고, 그 속에서 개선된 생각을 반영해야 한다. 하지만, 그렇다고 남들이 다 겪은 문제를 다시 겪을 필요는 없다. 이미 충분한 경험은 다양한 형태로 축적되어 있고, 이제는 그것들을 초보 개발자들이 이해할 수

있는 형태로 주어지면 되는 것이다. 다른 공학 분야와 마찬가지로 지식은 항상 기초적인 토대를 바탕으로 쌓아 올라가는 방향으로 발전한다. 기초적인 지식이란 실무에서 겪는 문제들에 대한 해답 들이며, 그런 해답들 위에서 이론적인 토대가 마련될 수 있다. 그리고, 다시 그런 이론적인 바탕에서 실무적인 개선이 이루어지며, 그렇게 쌓여진 것들이 다시 다른 형태로 사용자에게 다가갈 수 있는 제품 형태로 만들어 진다. 어쨌든 기초가 가장 중요한 것 임은 부인할 수 없다. 기초가 없으면 다시 허물고 쌓아올리게 되는 일이 잦아지게 되며, 다시 새로운 형태의 기준에 맞게 시행착오를 경험할 것이다.

시중에 있는 다양한 책들은 아직 시행착오를 겪었던 것을 충분히 반영하고 있지 못하며, 이런 지식들은 소위 발하는 "Guru"에게서만 듣거나, 혹은 그들이 쓴 책을 공부해서 얻을 수 있다. 하지만, 몸에 밴 습관은 쉽게 고쳐지지 않으며, 잘못된 지식은 현실에서 다시 다음 세대로 아무런 근거없이 전달된다. 왜냐하면, 그렇게 배운 사람들이 이미 중요한 직책에서 중요한 일을 하고 있기 때문이다. 그렇다고, 그들이 옳다는 것은 아니다.(또한, 그렇다고 그들이 잘못되었다고 이야기 하지도 않겠다.) 자신들이 경험한 것을 잘 정리하고 개선할 시간이 없었다고 생각되기 때문이다.

사실, 프로그램 개발자인 우리는 빨리 움직여야 하기 때문에, 천천히 기본적인 것들을 충분히 익힐 수 있는 시간이 없다. 왜 이런 현실이 주어 졌는지 근본 이유를 찾아 올라가더라도 새로운 것을 발견할 수는 없다. 즉, "코딩은 열심히 하고 많이 하지만, 제대로 하지 않는다."가 정답이다. 따라서, 이제는 제대로 하는 코딩에 대해서 이야기 할 때라고 생각하며, 그런 부족한 부분을 메워줄 수 있는 것들을 이 책에서 찾아보기를 기대한다.

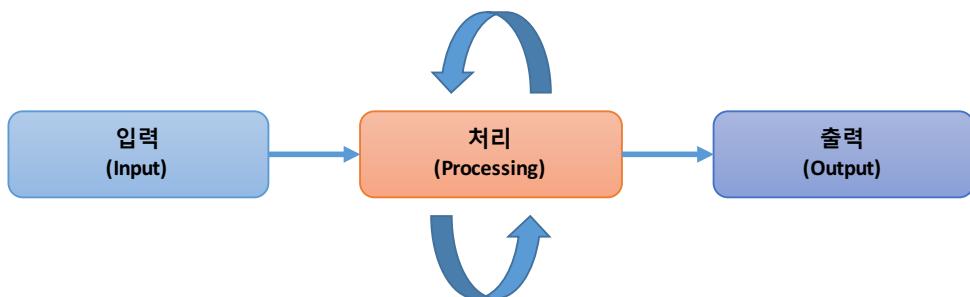
[프로그램을 처음 작성하는 사람들에게]

책으로만 코딩을 배운 사람들에게 처음 숙제를 주고 결과를 출력해서 가지고 오라고 하면 어떤 반응을 보일까? 아마, 상당히 힘든 일이라는 것을 금방 알 수 있을 것이다. 먼저, 어떤 프로그램을 설치해야 하고, 코드를 포함하는 파일은 어떻게 만들어야 하며, 코드 자체에 대한 구성문제, 코드 입력 후 컴파일은 어떻게 해야 하는지, 컴파일 오류가 발생했을 때는 어떻게 처리해야 하는지 등등 다양한 문제가 찾을 것이다. 이 말은 코딩은 책으로 지식을 습득하는 것 만이 전부가 아니라는 뜻이다.

코딩을 배우기 위해서는 일단 코드를 입력해 볼 수 있어야 한다. 그리고, 당연하겠지만 컴파일과 그 과정에서의 오류를 수정하는 연습, 그리고 실행 결과를 확인하는 과정 등이 필요하다. 정말 중요한 것은 숙제가 주어졌을 때 어디서부터 코딩을 해야할지, 어떻게 코드를 만들지 막막하다. 이때 접근할 수 있는 방법은 일단은 다음과 같은 세가지 부분으로 나눠서 생각하라는 것이다.

1. 입력으로 주어질 수 있는 것들에는 무엇이 있는가?
2. 출력으로 무엇을 보여주어야 하는가?
3. 그리고, 어떤 처리가 필요한가?

일단은 위의 2가지 정도만 가상으로 만들어서 제대로 되는지를 확인하는 것이 필요하다. 사실 입력과 출력, 중간 조작(처리)이 코딩의 가장 간단한 형태다. 쉽게 시작하기 위해서는 간단한 함수들을 응용할 필요가 있을 것이다. 입력에는 "scanf()"와 같은 함수를, 출력에는 "printf()"와 같은 함수면 충분할 것이다.



이제는 중간 조작을 어떻게 처리할 것인가를 고민하는 과정이다. 먼저, 중간에 해야할 일이 무엇인지 한 문장으로 써보자. 예를 들어, 팩토리얼(Factorial)를 구하는 방법이라면 아래와 같이 적어볼 수 있을 것이다.

"Factorial을 구한다."

이제는 앞에서 정한 입력을 이용해서 조금 더 세밀하게 적어보도록 한다. 이때 숫자의 변환이나 상세한 문제는 일단 생략하도록 한다.

"입력으로 받은 x라는 값을 이용해서, Factorial을 구한다."

이번에는 앞에서 정한 출력을 이용해서 조금 더 발전 시켜보도록 하자. 이때도 마찬가지로 상세한 문제는 아직 생각하지 않도록 한다.

"입력으로 받은 x라는 값을 이용해서, Factorial을 구하고, 이를 출력으로 전달한다."

이제는 이것을 책에서 배운 함수의 형태도 정의할 수 있는 수준까지 왔다는 것을 발견할 것이다. 즉, 입력과 출력이 있는 함수의 구현이 가능하다는 것이다. 이때, 얻어야 할 입력의 형태(Type)를 정해야 하는데, 함수가 원하는 입력을 정의할 수 있으면 된다. 그리고, 출력도 마찬가지로 출력을 위해서 형태를 정의하면 될 것이다. 다음과 같이 될 것이다.

```
unsigned int calculate_factorial(unsigned int nth) {
    return 0;
}
```

일단은 빈 함수 부터 시작한다. 아무 것도 없어도 된다. 그리고, 입력에서 사용자의 입력을 지금 막 정한 함수의 입력으로 사용할 수 있도록 변화하는 과정이 필요하다면, 입력 부분을 수정하도록 한다. 아래와 같을 것이다.

```
unsigned int user_input = 0;
...
printf("User Input? \n");
scanf("%d", &user_input );
```

출력도 포함해서 결과를 낼 수 있도록 만들기 위해서는 함수의 호출 결과를 얻어낼 필요가 있다는 것을 알 수 있기에, 다음과 같이 코드를 작성할 수 있을 것이다.

```
printf("The result : %d\n", calculate_factorial( user_input ));
```

그리고, 한번 실행해 본다. 어떤 결과가 나오는지 확인한 후에 제대로 결과가 나오지 않는다면, 일단은 입력과 출력에 문제가 있다는 말이기에 여기서부터 올바른 결과가 나올 때까지 고치는 과정을 반복한다. 만약 결과가 제대로 나온다고 생각된다면, 이제는 정말 중요한 "해야할 일"을 구현해야 할 것이다.

```
#include <stdio.h>
#include <stdlib.h>

unsigned int calculate_factorial(unsigned int nth) {
    return 0;
}
```

```

int main(void) {
    unsigned int user_input = 0;

    printf("User Input? \n");
    scanf("%d", &user_input);
    printf("The User Input Value : %d\n", user_input);
    printf("The Result : %d\n", calculate_factorial(user_input));

    return EXIT_SUCCESS;
}

```

지금까지 만들었던 프로그램을 정리하면 대략 위와 같이 만들었다는 것을 알 수 있을 것이다. 이제 입력과 출력에 대한 부분은 신경쓰지 말고, 정말 구현해야 할 로직(Logic)을 만들어 나갈 준비가 된 것이다.

이제 문제 자체로 돌아와서 "알고리즘적인 것들을 처리하자." "N!"과 같은 팩토리얼의 값을 구하는 것은 "1부터 시작해서 N까지 모든 숫자를 곱하는 것"이다. 입력이 얼마나 주어질지 모르는 상황에서 무턱대고 "1"부터 곱할 수는 없다. 따라서, 일단은 작은 수부터 주어서 어떻게 처리되는지 확인해보는 것이 좋다.

```

#include <assert.h>
...
assert(1 == calculate_factorial( 1 ));

```

"assert()"라는 함수는 "("내부의 값이 True인지 False인지를 따져서 확인해주는 역할을 한다. 최소 "1"이라는 입력에 대해서는 당연히 "1"을 팩토리얼 값으로 주어야 하지만, 아직 구현된 부분이 없기에 0을 돌려주어 실행이 멈출 것이다.

이제는 팩토리얼을 계산하는 함수를 조금 바꾸어서 "1"이라는 값을 돌려주도록 바꾼다. "return 1"정도면 충분할 것이다. 다시 추가적인 테스트로 이번에는 2를 넣도록 해보자.

```
assert(2 == calculate_factorial(2));
```

당연히 실행이 다시 멈출 것이다. 따라서, 이제는 더 이상 단순히 "return"문을 가지고는 힘들다는 것을 알 수 있을 것이다. 다음과 같이 수정해 보자.

```

unsigned int calculate_factorial( unsigned int nth ) {
    if ( nth == 1 )
        return 1;
    return 2;
}

```

앞에서 만든 assert들은 통과 하겠지만, 추가적인 "assert(6 == calculate_factorial(3));"과 같은 것은 통과하지 못할 것이다. 물론, 우린 이미 팩토리얼이 어떻게 계산 되는지에 대한 방법은 알고 있다. 이제는 그것을 구체적으로 코드로 표현하는 일이 남은 것이다.(지금까지 극히 코딩을 단순화 시켜서 진행하고 있지만, 이런 과정을 통해서 개발하는 것이 더 효과적인 경우도 있다.)

팩토리얼은 " $1 \times 2 \times \dots \times N$ "으로 계산된다. 따라서, 입력에 값 만큼 증가시키면서 순환적으로 곱셈을 해주어야 한다. 이 말은 중간 결과값을 저장할 변수가 필요하며, 그것을 "result"라고 하자.

```

unsigned int calculate_factorial(unsigned int nth) {
    unsigned int result = 1;

```

```

for (unsigned int i = 1; i <= nth; i++) {
    result = result * i;
}

return result;
}

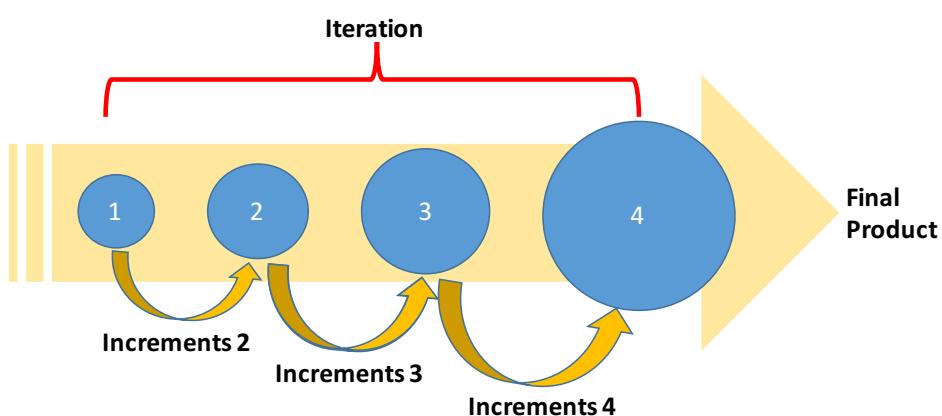
```

약간의 논리적인 비약은 있지만, 순환이 루프(Loop)라는 것과 증가되는 인덱스(Index)를 이용해서 결과값에 지속적으로 곱셈을 한다는 것 자체는 이해할 수 있을 것이다. 그리고, 그 결과값을 임시 변수에 저장해서 나중에 돌려준다는 것도 알 수 있을 것이다.

이제는 "assert()"들을 더 추가해서 올바른 값이 나오는지를 점검해 보자. 제대로 동작한다면 더 큰 값을 넣어도 동작할 것이다. 물론, 여기서는 방어적인 코딩이나 아주 큰 팩토리얼 값 까지도 구하는 것등의 과정은 생략되었다. 그것은 더 많은 알고리즘적인 고안을 통해서 가능할 것이다(방어적인 코딩은 입력 값을 어느 한정된 값을 낮출수도 있다. 큰 수의 팩토리얼 값을 구하는 것은 변수의 크기를 넘어서는 일이 되기 때문에 오버플로를 발생시킬 수 있다. 이때는 부분들을 다시 나누어서 계산하는 방법등을 사용할 수도 있을 것이다.).

다시 말이 길어지기는 했지만, 요지는 입력과 출력을 분리해서 생각하고, 구현해야 할 핵심논리에 맞게 입력을 변형하는 것과 출력을 위한 변환도 필요하다는 것을 언급했다. 그리고, 핵심 로직(Logic)의 구현에 있어서는 테스트와 함께 작성하는 것이 좋다는 예도 보여주었다. 이런 것들을 연습하는 이유는, 나중에 정말 어려운 숙제가 주어졌을 때, 한번에 작성하려고 시도하지 말고 조금씩 자주 확인하라고 이야기하고 싶기 때문이다.

프로그램은 절대 한번에 많이 작성하지 않아야 한다. 개인적인 실패 경험을 이야기 하자면, 학교를 다닐 때 멋 모르고 수천 라인을 작성한 후에 컴파일 문제를 잡고 실행해 보려고 했다. 물론, 실패했고 한동안 어떻게 해결해야 할지를 고민해야 했었고, 결국 그 문제로 인해서 결국 제대로 과제를 제출하지 못했었다. 경험적으로 봤을 때, 한번에 많이 작성하는 것은 과제의 실패로 이어질 가능성이 높다. 문제를 크게 만든 후에 푸는 것은 문제를 복잡하게 만들어서 풀이를 만드는 것과 같기 때문이다.



조금 설계하고, 조금 작성하고, 조금 실행하는 것이 중요하다. 중간 중간에 작성한 코드들이 제대로 동작하는지 끊임없이 확인해야 한다(단위 테스트를 하는 것도 이것 때문이다.). 중간 단계의 오류 사라지면, 통합에서 발생할 오류의 가능성도 낮출 수 있다. 또한, 버그를 찾는 것도 더 편해진다. 큰 기능을 완성해야 할 필요가 있다고 생각될 때도, 그 기능을 잘게 나누어 조금씩 작성하는 것이 좋다. 대략 하나의 함수를 만들었다고 생각되면, 반드시 그 함수를 실행해서 제대로 동작 하는지 확인할 수 있어야 한다.

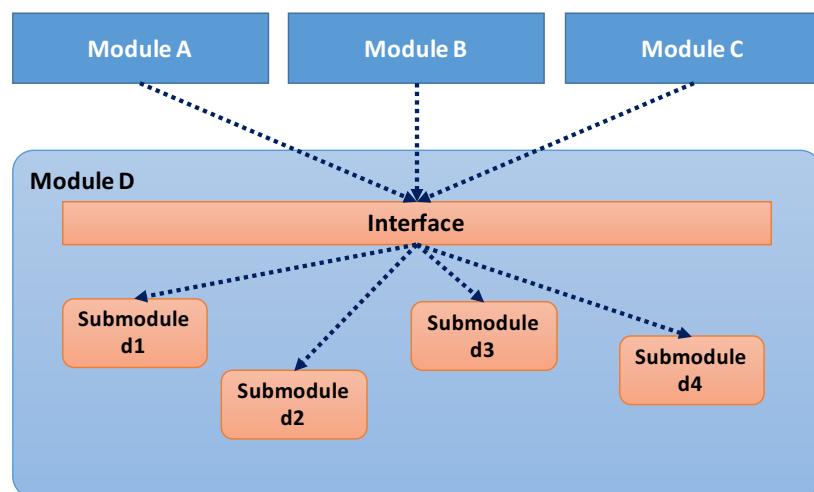
최근의 개발방법론 중에 "TDD(Test Driven Development)"와 같은 것이 있는데, 이 방법에서는 한 라인이나 조금만 변경이 있어도, 테스트를 실행할 수 있는 방법을 알려준다. 그리고, 그렇게 하는 것이 가장 버그를 수정하는 비용이 싸다. 즉, 버그가 발생한 시간과 그것을 발견하는 시간 사이의 간격이 짧아질수록 비용이 절감되는 것이다. 어쨌든 TDD에서도 중간 중간에 꼭 실행을 통해서 제대로 동작 하는지를 확인하는 "피드백(Feedback)"과정을 중요시하고 있다.

사실 코딩이 어려운 것은 코딩을 안해봤기 때문이다. 그렇다면, 코딩을 많이 하는 것이 유일한 해결방법이다. 물론, 무슨 코딩을 해야하는지 궁금할 것이다. 일단은 책에 나오는 예제를 전부 손으로 입력해서 실행해 봐야 한다. 그냥 눈으로 하는 "코딩 공부"는 아무 의미없는 일이다. 시간이 지나면 금방 잊어버리기 때문이다. 그리고, 요즘 시중에 나오는 책들이 제공하는 예제를 그냥 돌려 봐도 코딩을 배우지는 못한다. 직접 코드를 입력해서 일일이 틀린 것을 수정하면서 실행해 봐야 한다. 그게 코딩을 배우는 가장 빠른 길이다. 코딩은 손과 발을 이용해서 몸으로 익히는 것이기 때문이다.

설계가 중요하다는 이야기는 들어봤을 것이다. 물론, 그렇다고 설계를 처음부터 잘 할 수 있는 것은 아니다. 만약, 그게 가능하다면 대부분의 프로그래머는 직업을 다른 것으로 바꿨을지도 모른다. 설계는 경험에서 나오는 것이 크기에, 코딩을 하면서 익힐 수 밖에 없다. 하지만, 그렇다고 무조건 코딩부터 하라는 말은 아니다. 코딩하기 위해서는 계획이 필요하다. 어디에 무엇을 두어야 할지, 어떻게 나누어야 할지를 정해야 한다. 사실 그것이 설계의 모든 것이다.

가장 먼저 해야할 일은 단계를 나누는 것이다. 그리고, 각 단계를 세분화하면서 계속 쪼개는 과정을 반복하는 것이다. 그러고나면, 대부분의 경우 동사와 명사가 남는다는 것을 볼 수 있다. 동사는 대부분 함수가 될 수 있는 대상이 되며, 명사들은 함수가 처리해야 할 자료구조나 변수가 될 가능성이 높다. 그리고 나서 해야할 일은 연관성이 있는 함수들과 자료구조들을 같이 모으는 과정이다. 서술과 세분화를 통해서 일의 절차를 깨닫고 시나리오를 쓴다면, 등장 인물들은 자료구조와 함수들의 묶음이 될 것이다. 따라서, 이제는 시나리오를 완성하기 위해서 각각의 등장인물들이 어떻게 상호 연결되어 이야기를 전개할지를 정해야 한다.

입력과 출력, 그리고 처리가 합쳐지면 프로그램이 완성된다. 대부분 중요한 과정은 "어떻게 처리할 것인가"에 달려있다. 일명 "비지니스 로직"이라는 것을 구현해야 한다. 세분화를 통해서 나누어진 역할을 가진 모듈들이 서로 협력하면서 일을 처리하는 것이 "일종의 객체지향적인 사고의 시작이다". 따라서, 그렇게 만들어진 시나리오와 관계를 구체화 시키는 일이 코딩이 될 것이다. 큰 틀을 만들었으니, 이젠 그것을 구체적인 코드로 하나씩 써 나가도록 한다. 여기서 중요한 것은 절대 구체적인 정보에 의존하지 말고, 항상 정의된 함수를 통해서만 각각의 구성요소(등장 인물)들이 연결되어야 한다는 점이다.



구체적인 것에 의존하지 않는 것은 중요한 기법이다. 구현에 의존하지 않고 추상화된 함수의 인터페이스를 반드시 사용해야 한다. 외부에 있는(다른 모듈에 속한) 자료구조를 직접 접근하는 일은 없어야 한다. 만약, 그것이 정말 성능을 낮출 것 같다면, 나중에 해당하는 부분만 직접 접근할 수 있도록 만들어주면 된다. 처음부터 성능을 위주로 코딩하는 것은 위험한 생각이다. 구체적인 것에 의존이 심할수록 코드는 변경에 취약하게 되기 때문이다. 그리고, 모든 코드는 언제든 바뀔 가능성이 있다는 것이 기본 가정이다.

요약하자면, 코딩을 처음하는 사람들은(혹은, 조금 해본 초급자들은) 절대 한번에 많이 구현 하려는 생각을 접어야 한다. 조금 구현하고 반복 실행해서 결과를 확인하는 것을 게을리해선 안된다. 그리고, 반드시 자료구조와 같은 것에 지나치게 의존적인 코딩을 해선 안된다. 자료구조를 다루는 함수들을 정의하고, 그것을 사용해서 해당 자료구조에 접근하도록 해야한다. 파일 외부로 공개 되는 정보는 최소화 시켜야 하며, 될 수 있으면 특정한 상황을 가정해서 코딩하지 않아야 한다. 방어적인 코딩과 인터페이스를 검증하는 코딩(함수의 호출 결과를 검사하고, 호출된 함수에서 제대로 호출 되었는지를 검사하는)을 해야 한다. 코드가 너무 길어진다고 생각된다면, 반드시 나누어 주어야 할 것이다. 나누는 기준에 대해서는 나중에 따로 살펴볼 것이다.

[작은 것을 잘 만들어야 한다.]

프로그램은 명령어로 이루어진 코드 들의 연속이다. 학교 다닐 때 학생들은 고수준의 개념들에 대해서는 많이 배운다. 하지만, 실제로 그것이 무엇을 의미하고 어떻게 구현되는지는 생각해 보지 않는 경향이 있다. 예를 들어, "파일이 뭐야?"라는 질문을 받는다면 어떻게 대답할 수 있을까? 만약, 자신이 한 문장으로 설명하지 못하고 장황하게 설명을 늘어놓고 있다는 생각이 든다면, 그것은 잘 모른다는 것을 나타낸다고 보면 된다. 설명할 수 있다는 것은 명쾌하게 짧은 한 문장으로 표현할 수 있다는 것이다.

파일의 정의는 "바이트(Byte)의 연속"이다. 즉, 파일의 내용은 바이트의 연속으로 채워져 있으며, 그것을 읽기 위해서는 바이트 들을 해석해야 한다는 의미다. 바이트 들의 연속에 의미를 부여하는 것은 응용 프로그램이 할 일이며, 운영체제의 입장에서는 그냥 바이트를 저장하고 읽는 일만 한다. 즉, 구현의 역할이 나누어져 있다는 점이다. 해석까지도 운영체제가 맡게되면 좋겠지만, 그렇게 만들기 위해서는 모든 어플리케이션이 동일한 의미를 지니는 데이터를 저장하고 읽는다는 것을 가정해야 할 것이다.



이처럼 우리가 평상시 일할 때 사용하는 어휘들이 정확히 무엇인지를 이해하는 것은 중요하다. 흔히 네트워크를 이용할 때, 소켓(Socket)이라는 것을 사용하지만, 소켓이 어떻게 구현되는지는 모른다. 포트(Port)가 대략적으로 무슨 일을하는지는 이해하지만, 그것이 왜 필요하며 어떻게 구현되는지는 모른다. 따라서, 우리 지식 수준을 알기 위해서는 기본적인 어휘들에 대한 명확한 개념과 그것을 구현하기 위한 노력까지도 알고 있어야 한다. 전문가가 되기 위해서는 단순히 고수준의 개념을 익숙하게 사용하는 것에 그치지 않고, 내부 구현에 대한 사항까지도 잘 알고 있어야 하는 것이다. 마치 모래성을 쌓아 올리듯이 만든 지식 체계는 근본이 흔들리면 전체가 영향을 받게 되는 것과 같다.

C언어의 구현에 대해서도 마찬가지다. 복잡하고 난해한 알고리즘을 이해하는 것도 중요하지만, 그것을 어떻게 구현하는 것이 좋은지도 고민해야 한다. 실체가 없는 알고리즘은 현실에 적용하지 못할 수도 있다. 그렇다면, 코딩에 있어서 가장 기본은 무엇일까? 물론, 여기서 말하는 기본은 전체 코딩 실력을 좌우 할 수 있을 만큼 큰 영향을 가질 수 있는 것을 말한다. 많은 사람들이 고민해봤을 것이라고 생각하지만, 계속된 물음으로 얻을 수 있는 것은 단순하지만 명확하다. C언어에 한정해서 생각한다면, 그것은 당연히 "함수(Function)"가 되어야 할 것이다.

함수는 C언어에서 독립된 실행을 할 수 있는 최소 단위다. 즉, 명령문은 독립적으로 실행할 수 없다. 하지만, 함수는 돌립적으로 다른 곳에서 재사용될 수 있으며 변경없이 실행할 수 있다. 분리된 파일로 관리할 수 있으며, 필요시 호출해서 사용할 수 있는 것이다. 운영체제에서의 가장 기본 단위는 무엇일까? 논리적으로는 다양한 부분들이 있겠지만, 간단히 예를 들면 스케줄링의 최소 단위를 기본으로 삼는다. 어떤 운영체제의 경우에는 스케줄링의 최소 단위가 "쓰레드(Thread)"가 되기도 하고, 혹은 "프로세스(Process)"가 되기도 한다. 어쨌든 가장 기본적인 단위에 대해서 파악하는 것이 나머지 체계를 세우는 기반이 된다.

함수에 대한 확고한 기반을 만들기 위해서는 어떻게 해야 할까? 먼저, 함수가 "좋은 이름"을 가질 수 있도록 해줘야 한다. 함수가 좋은 이름을 가진다는 의미는 "함수가 하는 일이 그 이름으로 대표된다"는 뜻이다. 즉, 그 함수의 역할을 한정하는 이름을 가져야 한다. 될 수 있으면 정확한 이름을 줘야 한다. 따라서, 하는 일도 명확해야 한다는 것은 당연한 결과다. 함수의 이름을 통해서 사람들은 그 함수가 하는 역할에 대한 추상화(상상)를 진행할 수 있다. 즉, 구체적인 것을 모르더라도 그 함수를 사용하는데 지장이 없다.

함수가 좋은 이름을 가지기 위해서는 하는 일이 적을수록 유리하다. 이것은 단일하고 고유한 한 가지 역할에 충실한 함수를 만드는 것이다. 함수의 길이가 길어지거나, 혹은 함수의 파라미터들이 많아진다는 것은, 결국 그 함수가 하는 일이 많아진다는 의미로 받아들일 수 있다. 그렇지 않으면 함수가 길어질 이유가 없다. 물론, 어떤 함수의 경우에는 자신이 맡은 일을 처리하기 위해서 긴 라인으로 구성될 수도 있다. 이때는 계층적으로 함수를 구성해서, 전체적으로 평평한 구조(Flat)한 구조로 여러 작은 함수들로 다시 나누어야 한다는 신호다. 따라서, 함수의 길이를 줄이고, 계층화 시키고, 각각의 함수들이 사용하는 파라미터의 적정 숫자를 제한하는 것은 중요한 일이다.

함수가 외부의 구체적인 것에 의존하게 되면, 그것으로부터 독립적으로 실행될 수 없다. 따라서, 의존하는 부분을 최소화 시키는 것은 함수의 독립성(Independence)을 키우는데 중요하다. 구체적인 것에 의존하는 코드를 짜는 것은 결국 함수가 별도로 분리되어 관리되지 못하게 만든다. 만약, 의존할 것이 있다면, 적어도 동일 모듈의 수준에서 처리되어야 할 것이다. 물론, 더 좋은 것은 의존하는 부분이 함수가 정의된 파일로 한정시키는 것이 좋다. 함수가 다른 모듈에 정의된 함수를 호출하는 의존성은 어쩔 수 없이 존재한다. 하지만, 이때도 중요한 것은 의존하고 있는 함수의 구체적인 부분까지도 알고 사용하지 말아야 한다는 것이다. 즉, 의존하고 있는 함수의 내부적인 구현을 모르게 짜야 한다는 것이다.

함수의 파라미터 갯수는 적을수록 좋으며 4개를 넘어서는 안된다. 만약, 많은 수의 파라미터를 넘겨주어야 한다면, 그런 파라미터들을 묶어서 구조체(Structure)로 만들어 두는 편이 좋다. 즉, 함께 사용되는 데이터들은 함께 관리되어야 한다. 따라서, 파라미터는 그 구조체에 대한 포인터와 같은 형태가 될 것이다. 함수의 파라미터 수가 늘어나면 함수가 하는 일이 구체적이고 복잡해질 가능성이 높다. 즉, 추상화가 낮아진다(깨진다)는 말이다. 코딩의 핵심원리(사고방식의 원리도 마찬가지만)인 추상화가 약해지면, 코드는 쉽게 깨질 가능성이 높아진다. 물론, 성능 측면에서의 일부 희생이 있지만, 성능은 구조화된 코드(의존성이 높지 않은 코드)에서 더 쉽게 개선하기 쉽다는 점을 기억해야 한다.

함수의 이름(Ex, WriteData());

함수 내부의 코드들은 함수의 이름보다
한 단계 낮은 추상적인 의미를 가져야 한다.
(Ex, Write_Name(), Write_Age(), Write_Weight())

함수 내부의 구현 수준은 함수의 추상화 수준보다 하나 아래 수준으로 맞추는 것이 정답이다. 하지만, 그렇게 만들기는 쉽지 않다. 예를 들어, 피보나치 수열을 구하는 코드를 짜야 한다면, 첫 번째 수준은 "find_fibonacci_number()"와 같은 함수의 이름이 될 것이다. 내부로 들어가면, "read_input_from_user()", "calculate_fibonacci_number()", "return_output_to_user()"와 같은 함수들이 나올 것이다. 각각을 좀 더 들어가면, 사용자로부터 입력을 읽는 부분이 있고(read_stdin()), 읽어진 변수의 입력 값이 정확한지도 확인하고, 다시 그것을 이용해서 피보나치 수열을 구하고, 결국 파일이나 화면에 표시하는 코드들이 나올 것이다. 즉, 점점 더 추상화 수준이 낮아지면서 구체적인 구현으로 코드가 변하게 된다. 따라서, 한 함수의 내부는 일정 수준의 추상화에 고르게 만족되어야 하는 코드들로 구성된다. 만약, 추상화 수준이 들쑥 날쑥한 함수의 내부 구현이 있다면, 구체적인 부분들을 다시 분리해서 새로운 함수로 만드는 것을 고려하는 것이 좋을 것이다. 함수의 내부 구현은 함수의 이름이 지시하는 추상화 수준보도 하나 더 낮아야 한다.

함수 내부의 조건문 분기도 없으면 없을수록 좋다. 즉, 조건문이 아무 것도 없으면 그냥 쭉 내려가면서 실행하면 된다. 테스트하기도 편하다. 모든 입력에 대해서 처리가 간단하기 때문이다. 이런 상황이라면 함수의 내부에 구현된 코드가 실행되지 않는 경우는 없다. 따라서, 조건문이 복잡하면 함수의 테스트도 까다로우며, 복잡해지는 경향이 생긴다. 최대 3단계 이상의 중첩된 조건문이 같이 사용되는 경우를 막고, 복잡한 조건문은 분리된 함수로 처리하는 등의 노력을 해주면, 코드는 이해하기 쉽고 간단해 지게 될 것이다.

함수 내부에서 사용하는 변수의 개수도 제한하는 것이 좋다. 사용하는 변수가 많아지면, 복잡한 일을 하고 있다고 볼 수 있다. 이때는 7+2라는 법칙을 사용해 볼 수 있다. 즉, 5개에서 9개사이의 변수들만 사용하는 것이 좋다. 그 이상의 변수는 한 번에 사람이 머리속에 저장하기 힘들어 코드를 읽기 어렵게 만들 수 있다. 그리고, 가능한 변수들은 사용되는 곳과 가까운 곳에 정의하는 것이 일반적으로 더 코드를 읽기 쉽게 만든다. 일률적으로 함수의 상단부에 정의하는 것은 옛날 스타일이다. 그리고, 그랬을 경우 사용되지 않는 변수가 발생할 수 있다. 당연히 지역 변수는 읽기 전에 반드시 초기화하는 것이 원칙이다.

함수를 사용하는 측에서는 함수가 복귀값(Return Value)을 가질 때는 반드시 확인해야 한다. 실제로 종종 확인하지 않는 경우가 있으며, 이것이 버그의 원인이 되기도 한다. 수 없이 실행해도 거의 오류가 없는 경우도 있지만, 간혹 발생하는 오류를 점검하지 않아서, 시스템에 치명적인 영향을 주게되는 것이다. 특히, 운영체제와 같은 곳에서 서비스하는 함수를 호출한 경우에는 반드시 복귀값을 확인해야 한다. 예를 들어, 메모리를 할당하거나 파일을 생성하는 등등의 함수가 오류를 자주 발생시키지는 않지만, 사용할 메모리가 부족하거나 같은 이름의 파일이 존재하는 경우 등, 다양한 생각하지 못하는 경우에 대해서 오류로 연결될 수 있다. 복귀값이 있다는 뜻은 그것을 확인해야 한다는 의미로 해석해야 한다.



함수를 사용하기 위해서는 헤더(Header) 파일이 필요하지만, 외부에서 호출될 필요가 있는 함수의 선언만 가지고 있어야 한다. 또한, 함수의 내부에서 처리되는 자료구조에 대한 타입 정보만 알려주어야 하며, 상세한 자료구조의 세부 구현은 감추는 것이 코드간의 의존성을 줄여준다. 파일 내부에서만 사용되는 함수들은 "static"이란 키워드(Keyword)로 선언해 주어야 한다. 즉, 파일 외부에서 접근하는 것을 근본적으로 차단해야 한다. 변수나 함수와 같은 것들은 정해진 범위에서만 사용하는 것이 원칙이다. 범위가 넓어지면 넓어질수록 오류를 발생시킬 가능성이 커진다. 함수의 내부 변수도 마찬가지다. 특정 블록(Block: "{}")에서만 사용된다면, 블록내에 정의하는 것이 좋다.

간단한 함수지만 생각보다 많은 부분을 세밀하게 다루어줘야 하는 것이 좋은 코드를 만드는 첫 걸음이다. 모든 코드는 작은 단위 들로 이루어지며, 작은 단위 들을 더 탄탄하게 만들수록 코드는 오류가 줄어든다. 큰 일은 작은 일들로 나누어(쪼개)지며, 각각의 작은 일을 섬세하게 해야 전체 일을 쉽게 마칠 수 있다. 일을 할 때는 가장 작게 나눌 수 있는 단위가 무엇인지를 생각하고, 그 단위를 엄격하게 관리하는 것이 중요하다. 함수의 길이를 짧게 만든다고 개발자의 창의력에 영향을 주는 것은 아니다. 이런 연습이 많아질 수록 더 좋은 코드를 만들 역량도 커지는 것이다. "무엇을 알고 있는가?"보다 "무엇을 할 수 있는가?"에 초점을 맞춰서, 실제로 어떻게 일을 하는 것이 좋은지를 스스로 터득해 나가야 할 것이다.

[코드의 품질이란?]

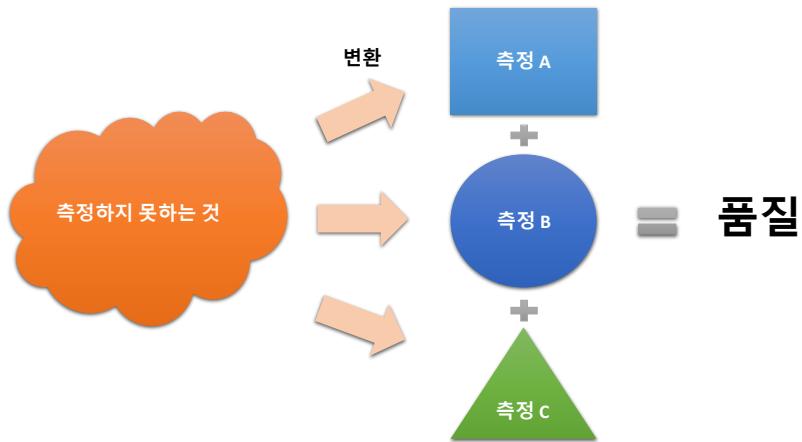
우리는 우리가 만드는 코드의 품질이 어떤지 알고 있는 것일까? 사실 품질이라고 하면, 거의 대부분 버그가 얼마나 있는지 만을 연상한다. 하지만, 품질에는 눈으로 확인할 수 있는 버그와 관련된 증상만 있는 것은 아니다. 사용자가 사용하면서 느끼는 품질을 외부 품질이라고 할 때, 내부의 개발자들이 느끼는 품질을 내부 품질이라고 볼 수 있다. 외부 품질을 수정하는데는 개발초기에 버그를 잡는 것보다 비용이 최대 260배(?) 정도 든다고 한다. 그리고, 내부 품질이 우수한 경우에는 외부 품질도 우수할 가능성이 높은게 일반적이다. 즉 코드의 품질이 높을 때, 사용자도 만족감을 느낄 가능성이 높다는 것이다. 물론 반드시 그런 것은 아니다. 단지 그럴 가능성이 높다는 점이다.

하지만, 개발 내부로 들어오면 이야기는 달라진다. 코드를 수정해야하는 프로그래머의 입장에서는 내부 품질이 더 중요한 것이다. 사용자의 요구사항이 빨리 바뀌고, 새로운 칩(Chip)이나 장비들이 지속적으로 등장 하는 때에는 변화를 수용하는 구조가 더 중요하게 된다. 이런 것들에 대해서 단순히 개발자에게 맡겨두기에는 뭔가 불안한 감이 있다. 그래서 활용하는 것이 측정 방법 들이며, 이를 도구화 시켜서 자동으로 언제든 품질 수준을 확인할 수 있도록 만드는 것이다.

대표적인 측정의 기본은 프로그램의 크기를 계산하는 것이다. 즉, 코드의 길이를 구하는 것이다. LOC(Line Of Code)라고도 불리며, 주로 1,000라인 단위로 KLOC라고 사용한다. 큰 프로젝트는 수 백만 라인에 달하기도 하고, 규모가 작다고 해도 수 천 라인일 경우가 많다. 이는 측정의 기본값이 되며, 만약 버그의 발생 빈도를 보고자 하면, LOC값으로 발생한 버그의 갯수를 나눈다. 이것이 버그 밀집도(Bug Density)와 같이 표현되는 값이다. 주로 PPM(Particle Per Million)단위로 사용되기도 하며, 다른 과제와 상대적인 크기를 비교할 때 사용된다. 주의 할 점은 코드의 중요도에 따라 발생하는 버그의 영

향도 차이가 있다는 점과 과제의 성격에 따라 코드의 크기가 영향을 받을 수 있다는 점이다.

코드에서 측정할 수 있는 것들에는 그 밖에도 무엇이 있을까? 예를 들어, 함수의 길이도 중요할 수 있다. 함수의 길이가 길다는 말은 함수가 많은 일을 할 가능성이 높으며, 함수내에 변수의 선언이나 다른 함수의 호출, 복잡한 조건문 등이 있을 가능성이 높다. 따라서, 가능한 함수의 LOC를 측정해서 줄이는 것도 품질에 영향을 주는 중요한 요소를 다룰 수 있게 만들어 준다. 함수의 파라미터 갯수를 세어서, 4개 이하로 만들도록 하는 것도 좋다. 함수의 길이에 대한 기준은 C의 경우에는 50라인 정도(어떤 사람은 100라인까지를 이야기 하기도 한다. 정해진 기준은 없지만 200라인을 초과하면 문제가 있다고 보는 것이 일반적이다. 참고로 리눅스의 경우에는 24라인을, 구글은 50라인을 권장한다.)를 보면 될 것이다.



앞에서 이야기 한 것과 마찬가지로 파라미터가 많아지면 그 함수를 사용하는 사람은 실수할 가능성이 높으며, 테스트 해야할 입력의 조합도 늘어나게 된다. 함수 역시 그런 파라미터 값들을 이용해서 해야할 일이 늘어나기에 복잡해지는 경향이 생긴다(Parameter들을 묶어서 구조체를 만들어 포인터로 넘기는 것은 상관없다.). 따라서, 파라미터의 개수는 코드의 복잡도에 직접적인 영향을 줄 수 있으며, 버그의 온상이 될 수 있다(함수의 내부보다는 함수의 호출 인터페이스에서 문제가 생기는 경우가 더 많음. 잘못된 파라미터나 함수의 사용법에 대한 오해 등의 이유로 인해서 발생한다.).

복제된 코드의 양을 측정하는 것도 좋다. 복제가 있다는 말은 변경이 발생했을 때 수정해 주어야 할 부분이 늘어난다는 것이다. 당연히 실수할 가능성도 증가하게 된다. 또한, 컴파일된 실행 이미지 자체도 늘어나기에 긍정적인 것보다는 부정적인 효과가 더 많다(복제된 코드를 함수로 만드는 것보다 속도는 빠를지도 모른다.). "DRY(Do Not Repeat Yourself)"라는 원칙이 바로 이것이다(물론, 다양한 상황에서도 사용되지만). 코드의 "Copy-and-Paste"는 개발자가 반드시 하지 말아야 할 것들 중에 하나이다. 하지만, 여기에서는 문제가 있다. 얼마나 많이 복사된 코드가 있어야지 문제로 볼 것인가이다. 대부분의 경우 10라인에서 5라인 사이의 복제된 코드가 있다면, 문제로 삼는 것이 좋을 것이다(어떤 툴에서는 6라인을 기준으로 하기도 한다.).

특정 디렉토리에 있는 코드들의 LOC합도 측정해볼 가치가 있다. 예를 들어, 특정 파일이나 디렉토리의 코드들이 지나치게 큰 LOC를 가진다면(물론, 여기서는 3rd Party에서 제공받은 것들은 제외함), 파일이나 디렉토리를 분리해야 할 가능성이 있다. 즉, 다양한 역할을 맡고 있 늘 가능성이 높다는 것이다. 코드는 고른 분포를 가질 때, 문제가 줄어드는 경향이 있다. 이 말은 역할을 세부적으로 나누고, 자신에게 주어진 역할의 범위 내에서만 책임을 가지고 구현하면 된다는 것이다.

"Assert()"와 같은 함수의 빈도를 조사하는 것도 좋은 방법이다. 즉, 전체 LOC에 대해서 얼마나 많은 "Assert()"를 사용하고 있는지를 보는 것이다. 왜냐하면, "Assert()"가 많을수록 버그가 줄어드는 경향이 있기 때문이다. 이 말은 개발자들이 예외 상황에 대해서 좀 더 꼼꼼하게 디버깅 했다는 다른 표현일 수 있다. 이때는 단위 테스트와 같은 것으로 "테스트 범위"를 LOC에서 측정해서 함께 사용하면, 더 높은 "

테스트 범위(Test Coverage)“를 얻을 가능성이 높다.

단위 테스트나 자동화 된 테스트를 하고 있다면, 당연히 "테스트 범위"를 항상 확인해야 한다. 어떤 테스트가 부족한지도 모른 상황에서 무한정 테스트를 늘리는 것은 효과적인 방법이 아니다. 따라서, 측정한 후에 보완하는 것이 좋다. 실제로 자신의 코드가 제대로 동작 하는지 테스트가 안되는 경우도 많다. 이런 것들은 특정 오류가 발생하는 빈도가 극히 낮거나, 혹은 방어적인 코드를 심어둔 경우, 또는 사용 되지 않는 코드 등일 경우가 있다. 각각에 대해서 테스트를 보완할 수 있는 조치를 취하거나 코드를 삭제해야 할 것이다.

여기서 보여주는 것들은 어렵지 않게 측정할 수 있는 것들이다. 그리고, 관련된 툴들도 이미 존재한다. 오픈 소스로 이용할 수 있는 것들이기에 비용을 들이지 않고도 할 수 있다. 문제가 되는 것은 개발자의 자세(Attitude)밖에 없다. 물론, 이런 것들이 사소하게 보이고, 별로 중요한 일이 아닐 것처럼 보일지도라도, 그런 작은 것들이 모여야 전체를 이룰 수 있다. 그리고, 개선이란 현재 상태를 모르는 상황에서는 불가능하다. 현재 우리가 제대로 일을 하고 있는지를 먼저 측정하고, 그 후에 어떻게 개선할 것인가를 스스로에게 질문해야 할 것이다.

[프로그래머]

프로그래머는 자신이 하는 일이 "남을 돋는 것"을 목표로 해야한다. 프로그래머는 자신이 만든 제품의 "품질"을 첫 번째 목표로 삼아야 하며, 품질 확보를 위한 노력에 최선을 다해야 한다. 남을 위한 것이 자신을 위한 것임을 깨닫고, 높은 품질을 추구하는 것이 프로그램의 전체 비용을 줄여준다는 사실을 알아야 할 것이다. 하지만, 결국 프로그램도 생산성을 따질 수 밖에 없다. 혼자서 재미 만을 추구하는 것도 좋지만, 팀으로 일할 때는 혼자만의 것이 되지 않는다. 코드를 작성한 사람은 자신이지만, 그 코드를 소유한 사람은 모두이기 때문이다.

코딩의 첫 번째 목표는 컴퓨터를 이해시키는 것이 아니라, 다른 개발자의 이해를 돋는 것이다. 자신이 작성한 코드지만, 읽는 것은 남들이 더 많이 한다. 아니라고 생각할지도 모르지만, 팀으로 하는 과제에서는 남들도 당신의 코드를 많이 본다. 문제가 발생하면 어디에서 발생했는지 당신도 찾아야 할지도 모르지만, 이것은 단순히 책임 소재를 따지자는 것이 아니라 효율의 문제다. 자신이 만든 코드를 자신이 이해하지 못하지는 않을 것이다. 하지만, 시간은 한정적이고 더 중요한 일에 사용 되어야 하기에 낭비할 필요는 없다.

코딩의 두 번째 목표는 품질을 높이는 것이다. 품질이란 검증 됨을 말한다. 검증되지 않은 코드는 품질이 보장된 코드가 아니다. 검증되지 않은 코드가 아무리 늘어나더라도 효율적인 개발은 되지 못한다. 결국, 검증 시간만 더할 뿐이다. 테스터는 코드의 문제를 발견하기 위해서 실행할 뿐이지, 코드의 버그까지 고쳐주지 않는다. 따라서, 결국 돌아오는 것은 개발자의 시간 부족이다. 가능한 자신의 손에서 마무리 지을 수 있는 일은 해야한다. 단위 테스트나 모듈 수준의 통합 테스트 정도는 개발자가 충분히 소화할 수 있는 부분이다.

코딩의 마지막 목표는 만족이다. 사용자를 만족시키는 것과 더불어, 스스로의 일에도 만족해야 한다. 자신이 만족하지 못한다면, 누구도 만족시키지 못한다. 스스로의 일에서 즐거움을 찾아내고 그것에 전념할 수 있는 사람이라면, 어떤 상황이 주어지더라도 목적에 충실할 수 있을 것이다. 스스로 만족하지 못하는 일이라면, 즐거움을 찾는데 두려워할 필요가 없다. 선택이라는 것은 언제나 내게 주어진 것이지, 남이 나의 인생을 조정할 수는 없다. 물론, 그런 능력이 그냥 주어지는 것은 아니다. 댓가는 삶의 조그만 변화를 꾸준히 유지하는 것이다.

프로그래머의 일은 혼자하는 것이 아니다. 아무리 작은 프로그램을 짜더라도 공개되는 위치에 선다면, 코드의 품질에 대해서 자신감을 가져야 한다. 이러한 자신감은 스스로가 부여하는 것도 좋지만, 남들의 피드백을 빨리 얻어도 된다. 골방에서 컴퓨터 앞에 담배와 콜라로 찌들어가는 인생을 사는것이 아니라, 넓은 세상에서 친구들을 사귀면서 깊이와 넓이를 더하는 삶을 살수 있다. 코드는 당신만이 사용하는 언

어가 아니며, 전 세계 모든 개발자들이 공통으로 사용하는 언어다. 그런 것을 익히는 사람이 닫혀진 사고를 가질 수는 없다. 선택은 스스로의 몫이지만 그것을 이용하는 사람들을 위해서 조금만 더 소통하려고 노력하는 것은 큰 차이를 만들어 낼 것이다.

[과제에서 살아남는 방법]

코딩은 다양한 활동이 동반된 일이다. 단순히 코드만 만들면 된다고 생각할지도 모르지만, 나름의 전략과 전술일 필요하다. 한 사람이 만들더라도 그런 것들은 필요하며, 적절한 순간에 적절한 도구가 사용되는 것이 중요하다. 전략은 일종의 과제에 대한 전체 밀그림이다. 전술은 그 밀그림을 어떻게 완성할 것인지 결정하는 일이다. 도구는 개발을 쉽게 할 수 있도록 사용 되는 것들이며, 좋은 도구는 생산성을 높이는데 많은 도움을 줄 수 있다. 물론, 좋은 도구를 가지고 있다고 좋은 코드를 만들고, 멋진 과제를 할 수 있다는 말은 아니다. 만족되어야 할 것들이 만족되지 않으면, 그런 것들은 거추장스러울 뿐이다.

코드는 절대 한번에 많이 작성하려고 해선 안된다. 코드의 변화량을 줄이고, 그 사이 사이에 코드에 대한 검증(Test)를 집어넣어야 한다. 제대로 동작하지 않는 코드를 많이 짠다고 높은 생산성을 보이는 것은 아니다. 예를 들어, 단위 시간당 작성되는 코드의 길이를 보면, 전체 과제 기간동안 하루에 대부분 10~20 라인 정도를 작성하는게 일반적이다. 물론, 그 정도도 많다고 하는 사람들도 있다. 대부분의 경우 5~10 라인 정도가 적당할 것이다. 이유는 과제에 대한 요구사항을 수집하고, 설계를 하고, 구현 및 테스트 등 등의 활동을 기간에 포함하기 때문이다. 여기에 유지보수하는 기간까지 포함하면 하루에 더 짧은 라인밖에 만들지 못한다.

생산성이 좋다는 말은 품질이 “우수한 제품을 저렴하고 빨리 만든다”는 말이다. 당연히 전제 조건은 품질이다. 즉, 코드의 품질을 높이 가져가지 않고서는 나머지 것들은 달성되기 어렵다. 달성하더라도 추가적인 비용이나 시간이 더 들어가서, 시장 선점이나 경쟁사를 추월하지는 못한다. 따라서, 코드는 자주 검증(Test)되고 검토(Review)되어야만 한다. 높은 생산성을 달성하는 조직의 특징은 그런 것들이 만족된 팀이다. 한번에 코드를 많이 작성하고, 나중에 그것을 테스트 하려는 것이 일반적인 개발자들의 생각이다. 그렇게 해서는 일반 개발자의 수준에만 머물게 된다. 더 높은 수준으로 올라가기 위해서는 코드 작성과 테스트의 간격을 최대한 줄여야만 한다.

단위 테스트(Unit Test)는 이런 것에 좋은 도구를 제공한다. 일반적으로 C언어의 단위 테스트의 최소 단위(Unit)는 함수다. 하나의 함수가 만들어지면 테스트 되어야 한다는 말이다. 물론, 선 코딩, 후 테스트 전략으로도 일단은 잘 할 수 있을 것이다. 조금 더 욕심을 낸다면, TDD(Test Driven Development)와 같은 것을 해볼 수도 있다. 요점은 코드가 만들어지는 시점에 코드를 검증하는 것이 가장 비용이 싸고, 가장 오류를 걸러낼 가능성이 높다는 것이다. 하나의 함수를 더 쪼개서 하나의 블록("{ }"으로 묶인)이 작성될 때마다, 테스트를 돌려보는 것도 좋다. 어쨌든, 함수 단위보다는 더 짧기 때문이다. 물론, 이때는 단위 테스트의 도움을 제한적으로 얻을 수 있을 것이다.

코드의 변화량을 줄여서 꾸준히 검증하는 것만이 제대로된 코드를 만들게 될 가능성을 높여준다. 단번에 코드를 많이 짜려고 하면, 반드시 실패하고 만다. 아니, 실패해서 시간이 더 많이 걸릴 것이다. 버그가 발생한 곳을 찾는데 걸리는 시간이 더 길어지며, 버그를 고치는데도 걸리는 시간도 길어지고, 코드의 수정이 미치는 파급효과를 검증하는데도 시간이 더 걸린다. 즉, 작은 부분이 수정되거나 추가되었을 때, 이를 검증할 수 있는 도구와 방법을 사용한다면, 버그 발생의 범위는 그 작은 수정이나 추가가 있었던 부분에 한정된다. 따라서, 그런 부분을 파악해서 고치는 것은 굳이 디버거(Debugger)를 사용하지 않아도 될 것이다. 범위가 늘어나면 사람의 주의도 분산되며, 논리의 한계도 드러나게 된다. 작은 범위로 한정하면 집중적으로 버그를 탐색할 수 있어, 더 효율(단위 시간당 찾은 버그의 개수)이 높아질 것이다.



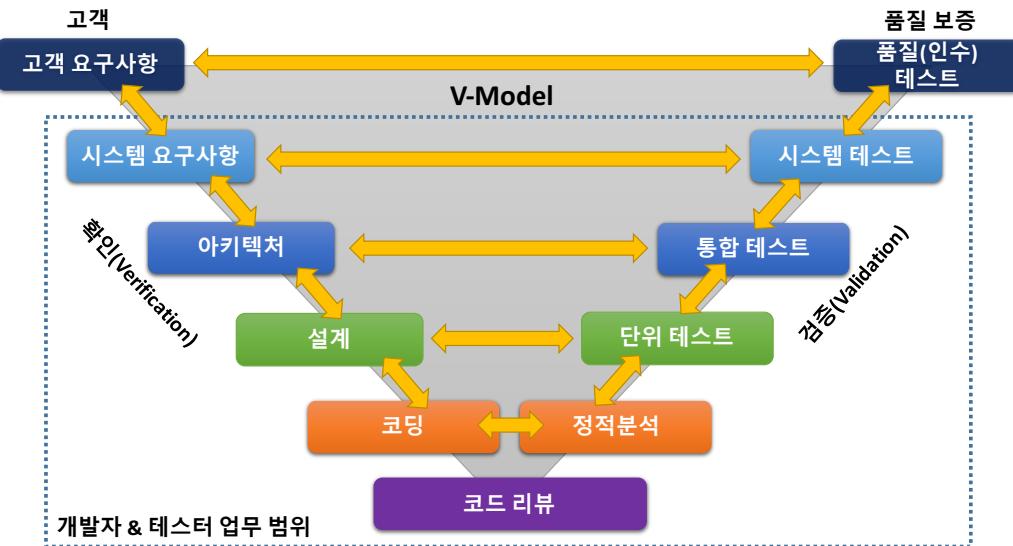
성공적인 과제는 큰 밑그림으로부터 시작한다. 그렇다고 이 그림이 실행에서 고쳐질 가능성이 없는 것은 아니다. 하지만, 큰 그림을 그리는 이유는 코드의 배치와 성능에 관련된 일이라 무시할 수 없다. 물리적인 코드의 배치가 중요한 것은 코드의 꼬임(Spaghetti Code)를 방지해줄 수 있기 때문이다. 구현하려는 기능에 대해서 어떤 모듈이 어떤 책임과 역할을 할지를 정해줄 수 있기 때문이다. 즉, 그렇게 배정받은 역할과 책임만을 충실히하는 코드를 만들 수 있는 틀(Frame)을 제공해 준다. 그 틀에 어긋나는 것은 검토의 대상이며, 다시 다른 모듈을 만들지, 아니면 기존의 모듈의 역할과 책임을 확대할지를 결정할 수 있게 된다. 어쨌든 전체적인 그림을 만드는 과정을 생략해선 안되며, 세세한 그림이 완성되어가면서 전체적인 그림도 일정 부분은 수정이 될 것이다.

코드를 작성하기 전에 코드에 사용될 스타일을 정하는 것은 코드를 읽는데 도움이 되는 일관성을 제공해 준다. 코드를 만드는데 사용되는 규칙에는 스타일 표준이 있으며, 각각은 여러 사람이 만드는 과제에는 필수적으로 사용된다. 혼자서 만들더라도 보기 좋은 코드를 만드는 것은 중요하다. 나중에 남에게 공개될 코드라면 더욱 신경써야 한다. 스타일은 겉모습을 보여주는데 집중하는 것이라면, 표준은 코드의 사소한 오류를 방지해주기 위한 성격 강하다. 따라서, 읽는 것에도 도움이 되는 실수를 방지할 수 있는 기법을 만들어서 사용하면 된다. 이미 이런 부분들은 인터넷에서 쉽게 구할 수 있을 것이다. 그런 것들을 이용해서 자신만의 코딩 규정을 만들면 된다.

코딩 룰(Coding Rule)을 만드는 것은 프로그래머의 창의성을 제약하는 것이 아니다. 무엇을 어떻게 할지는 프로그래머가 창의성을 발휘해야 하는 부분이고, 그것을 코드로 옮겨서 표현하는 것은 창의성의 제약이 아닌, 현실 세계에서의 구현에 대한 것이다. C문법이 만들어 졌다고, 사람의 창의력이 제약을 받았다고 할 수 있을까? C언어를 사용하는 규칙을 만들었다고 프로그래머가 자신의 생각을 C언어로 표현하지 못할까? 규정은 단순히 "그림의 액자"와 같은 구실을 한다. 그림의 내용은 화가가 자신이 생각하는 바를 표현한 것이다. 액자가 필요한 것은 그림을 더 잘보이게 걸어두기 위함이지, 그림의 내용을 제약하기 위한 것이 아니라는 것을 알아야 한다. 코딩 룰에 대한 반대가 많은 이유는 자신의 익숙함을 버려야 한다는 거부감 때문이지, 그것이 옳지 않아서가 아니다.

과제는 작은 반복과 그보다 조금 더 큰 반복, 그리고, 데모와 같은 더 큰 반복의 결과로 만들어진다. 각각의 반복 사이에는 단위 테스트, 통합테스트, 시스템 테스트와 같은 것들이 끼어 있으며, 반복의 결과물에 대한 검증은 “함수->모듈->서브 시스템->시스템”으로 이어지는 과정의 연속이다. 여기에 더 필요한 것은 다양한 시각이 포함되어 있어야 한다. 즉, 다른 사람의 시각으로 바라본 코드에 대한 개선방안이 덧붙는다면, 대부분의 버그는 제거될 가능성이 높다. 코드 리뷰는 그 다양한 시각을 갖추는데 반드시 필요한 활동이며 객관적으로 측정하기는 어렵다. 객관적인 기준이 될 수 없기에, 좀더 객관화가 필요한 부분이다. 즉, 무엇을 사람이 검사할지, 어떤 부분을 중점적으로 검토할지를 미리 정하고, 그것을 기준으로 코드 리뷰를 하는 것이 도움이 될 것이다. 개인에 대한 비판을 넘어서기 위해서는 이런 객관화된 리스트를 만들어서 사용해야 할 것이다.

프로그램 개발에 있어서 데모과정은 반드시 필요하다. 내부 데모도 필요하며, 고객을 대상으로 한 외부 데모도 필요하다. 이를 통해서 얻을 수 있는 것은 그것을 사용하는 사람들의 피드백이며, 과제의 완료쯤에 얻는 피드백보다는 훨씬 가치가 높다. 즉, 데모를 마지막에 한 번만 한다면, 피드백 내용은 결과물에 반영되지 않을 가능성이 높다. 이미 제대로 동작하고 있다고 생각되는 제품에, 추가적인 변경을 가해서 위험한 상태로 만들 이유는 없는 것이다. 하지만, 그런 제품은 이미 시장에서 경쟁력이 없다는 것은 금방 눈치챌 것이다. 따라서, 중간 중간에 데모 과정을 넣어두는 것은 반드시 필요하다. 중요한 마일스톤(Milestone)마다 그런 데모들이 있어야 과제의 진척도도 객관적으로 파악할 수 있다. 빨리 얻을 수 있는 피드백은 그것이 반영될 기회를 높일 수 있으며, 그것을 통해서 좀 더 고객이 원하는 제품을 만들어낼 가능성을 높이게 된다.



작성된 코드는 정확히 원하는 동작을 수행할 수 있어야 완료된 것이다. 물론, 이 상태라고 해도 100%완료라고 부르기 힘들다. 왜냐하면 나중에 변경될 가능성이 있기 때문이다. 개발자가 말하는 “90% 구현”되었다는 말은 아무것도 구현되지 않았다는 것과 다르지 않다. 따라서, 누군가가 일을 90%했다고 말한다면, 그것은 제대로 되지 않았을 가능성이 더 높은 것이다. 소프트웨어 개발에서는 그런 말들을 개발자들이 많이 하지만, 대부분의 경우 90%완료는 나머지 10%를 완료하기 위해서 지금까지 들어간 노력이 또 들어갈 것이라는 말로 들어야 한다. 따라서, 아직 가야할 길이 멀다는 뜻이다.

소프트웨어 개발과제는 기능의 완료는 반드시 "All-or-Nothing"으로 생각해 한다. 몇 %구현이란 없다. 물론, 기능 구현율을 말할 때는 구현율이 몇 %인지를 말할 수도 있을 것이다. 이때도, 구현된 기능들은 전부 검증된 상태라는 것은 확신할 수 없다.

과제를 성공적으로 수행하기 위해서는 작은 반복을 꾸준히 해야함과 더불어, 검증이 반복의 사이에 꼭 포함되어 한다. 될 수 있으면 작은 변화를 축적하는 방식으로 개발되어야 하며, 검증은 항상 코딩과 같이 동시에 행해지는 활동으로 보아야 한다. 빨리 검증 할수록 버그의 파급효과는 줄어들게 되며, 도구를 사용할 수 있다면 그런 것들도 과제 초반부터 적용하는 것이 좋다. 코딩 룰과 코드 리뷰, 단위 테스트 프레임워크(Framework) 등은 버그를 줄여주는 훌륭한 도구와 방법 들이며, 그런 것들을 동원해서 양질의 코드를 생산하는 것을 목표로 해야한다.

작성된 코드들은 제대로 동작하는지 사람들에게 보여줄 수 있어야하며, 이를 위해서는 배포와 빌드(Build)를 함께할 수 있는 도구도 필요하다. 개발자의 창의력이 이와 같은 틀 속에서도 유지될 수 있는 이유는, 도구는 생각을 형상화하고 구체화 하는데 도움이 되지, 도구로 인해서 제약받는 개발은 없다는 것이다. 도구를 잘 못 사용하는 경우는 도구의 목적이 아니라 사람의 의도가 개입되기 때문이다.

[참고] “Software Project Survival Guide - Steve McConnell”는 과제의 책임을 맡은 사람이라면 반드시 한번은 읽어봐야 할 책이다. 힘든 과제에서 어떻게 살아남을 것인지를 진지하게 고민한 해결책을 발견할 수 있을 것이다.

[테스트의 중요성]

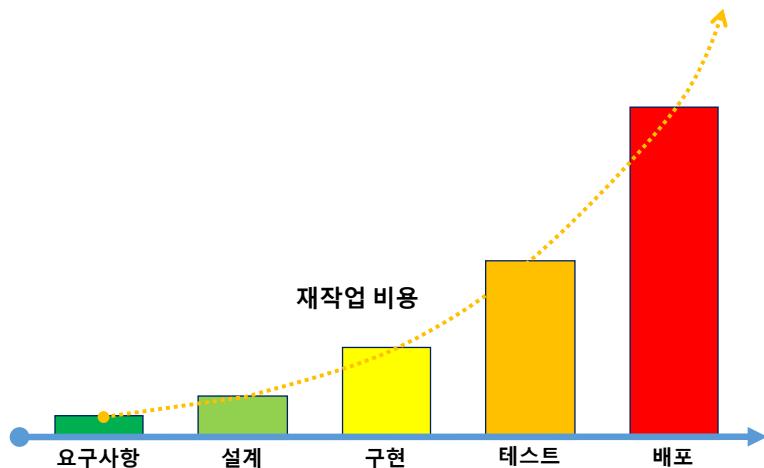
소프트웨어 개발자가 알아야 할 지식에는 “문제를 해결하려고 시도하는 도메인”, “특정 언어를 사용해서 구현”, “구현된 것을 검증하는 방법” 등이 있다. 문제를 해결하는 도메인에 대한 지식은 주로 일을 하는 과정에서 익히게 된다. 처음부터 도메인을 잘 알고 있으면 좋겠지만, 일은 마무리 될 때 가장 많은 지식을 소유하게 된다. 언어를 사용해서 구현하는 부분은 선행할 수 있는 부분이다. 즉, 특정 언어를 사용해

서 구현하는 연습을 하는 것으로 익힐 수 있다. 가장 문제가 되는 부분은 "테스트를 어떻게 하는가"와 관련된 "검증"이다.

학교에서는 검증에 관련된 부분은 제대로 다루지 않는다. 대부분 학교 숙제는 결과물이 제대로 출력되는지를 보지만, 다양한 입력이나 조건에 대해서는 어떤 결과가 나오는지 검사하지 않는다. 따라서, 학생들이 배울 수 있는 것은 도메인의 이해와 해결 방법의 구현에 한정된다. 사실상 테스트는 회사에 들어와서 이런 저런 일을 하면서 스스로 익히는 것이 일반적이다. 문제는 여기서 발생한다. 개발자의 테스트는 단순하며 문제를 제대로 검출해 낼 가능성이 낮다는 점이다.

"개발자가 검증도 해야하나?"라고 질문한다면, "당연하다"라는 것이 대답이다. 개발자는 자신이 코딩한 것을 스스로 증명할 수 있어야 한다. 물론 세상에 완벽한 소프트웨어는 존재하지 않는다. 하지만, 적어도 가능한 버그를 줄이고 출시해야하는 것이 개발자의 역할이다. 코딩을 끝냈다고 일이 완료가 되는 것이 아니라, 코드가 제대로 동작함도 보장해야 한다. 즉, 원하는 기능이 제대로 구현되어 있으며, 다양한 허가된 입력 조건에서 버그가 발생해서는 안된다.

개발자는 자신의 코드에 대한 취약점과 동작하는 조건을 잘 알고 있다. 하지만, 사용자는 그런 것들을 모른다. 따라서, 버그가 발생했을 때의 책임은 당연히 개발자의 몫이다. 테스터는 버그가 없음을 증명하는 사람이 아니라, 버그가 있는지 확인하기 위해서 프로그램을 실행하는 사람이다. 그들의 관점은 더 많은 버그를 찾는데 있다. 그리고, 개발자는 그렇게 찾아낸 버그를 검토해서 제거해야 할 의무가 있는 것이다. 하지만, 개발자가 테스트의 역할을 수행해야 할 수도 있다. 즉, 버그가 발생하는 시점이 버그를 찾아내고 제거하는데 가장 비용이 싼 순간이라는 것이다.



코드가 만들어진 시점과 버그를 제거하는 시점이 가능한 가까울수록 시스템은 안정적으로 동작하게 된다. 코드에서 버그를 줄이는 방법은 여러가지가 있지만, 대표적인 방법은 "단위 테스트"와 "코드 리뷰"다. 그리고, 이 두 가지가 가장 잘 안되는 것도 현실적인 문제다. 물론 둘 다 개발자의 몫인 부분이라는 점은 공통이다. 하지만, 단위 테스트는 코드를 작성하는 사람과 동일한 사람이 테스트를 실행한다는 것과 코드 리뷰는 다른 사람이 참여한다는 점이 다르다. 따라서, 개발자는 두 가지 기술을 같이 익혀야 버그를 최대한 줄일 수 있다.

단위 테스트는 단위 테스트를 실행하는 환경과 테스트 케이스, 검증 범위 결과 확인이라는 과정을 포함한다. 코드 리뷰는 코딩 룰과 스타일, 코드를 이해하기 쉽게 만드는 기법, 리뷰 툴과 참여자들이 필요하다. 사실 학교라는 상황에서 두 가지를 다 익히면 좋겠지만, 코드 리뷰는 다른 사람의 참여가 필요하다는 점에서, 오히려 단위 테스트가 비용 면에서는 좀 더 효과적일 수 있다(물론, 그렇다고 중요도가 낮다는 말은 아니다. 둘 다 조합해서 하는 것이 버그를 더 많이 제거할 수 있다. 일반적으로 일정한 형식을 갖춘 코드 리뷰가 더 많은 버그를 찾아낼 수 있다고 알려져 있다.).

학교에서 프로그램 숙제를 낸다면, 그것을 검증한 결과도 같이 요구해야 할 것이다. 즉, 테스트를 실행한 케이스들의 모음(물론, 실행할 수 있어야 한다.)과 테스트로 측정된 검증 범위 결과(이것은 리포트로 자동으로 생성할 수 있다.)를 같이 요구해야 한다. 단순히 실행 결과만을 위주로 한다면, 결코 좋은 코딩 습관을 가질 수 없다. 그리고, 정말 중요한 부분은 작성된 코드에 대한 "피드백"이다. 지식만 전달할 목적이라면 지금도 충분하다고 보지만, "성장"시켜주고 싶다고 생각한다면 적극적으로 코드에 대한 피드백을 주어야 한다. 그것이 힘들다면 차라리 툴을 이용해서 나온 문제점을 점수에 반영하는 방법을 사용할 수도 있을 것이다.

결과도 중요하지만 과정과 문제를 해결하기 위한 방법도 중요하다. 똑같은 결과물을 출력 했다고 해서, 더 나은 성장을 했다고 보장할 수는 없다. 따라서, 교육이 추구하는 부분이 "성장"이라면, 도전적인 목표와 함께 결과물(출력과 소스코드, 테스트 케이스 등)에 대한 피드백도 주어야 한다. 단순히 몇 점을 주었다고 피드백이 끝나는 것이 아니다. 이건 모든 교육에서 마찬가지다. 학생이 문제 풀이에 만능숙하고 그것의 원리와 풀이하는 과정에 대한 피드백이 없다면, 성장은 그 문제의 풀이(테두리 안)에서 멈출 것이다.

[“좋은 코드”가 가지는 특징]

프로그래밍할 때 좋은 코드가 가져야 할 특징을 이해하는 것은 중요하다. 코드는 컴파일된 이미지만 중요한 것이 아니라, 다른 사람과 같이 읽어야하는 산출물이기 때문에 관리를 꾸준히 해 주어야 한다. 좋은 코드란 "쉽게 이해가 되는 코드"다. 간단히 말하지만 굉장히 어려운 일이기도 하다. 즉, 자신의 시선으로 바라보아야 하는 것이 아니라, 타인의 시각으로 보아야 하기 때문이다. 하지만, 좋은 코드가 가져야하는 특징들을 지키다보면, 이런 것들은 어느새 몸에 익하게 될 것이다. 자신이 코딩을 잘하고 남들이 만들어 놓은 코드를 잘 이해한다고 생각하는 것은 좋다. 하지만, 남에 대한 배려가 코드에 묻어나는 사람이 정말 역량 높은 개발자다. 전자는 그냥 똑똑한 개발자일 뿐이지만, 후자는 지혜로운 개발자가 된다. 지혜로움은 항상 똑똑함을 넘어서게 된다는 것을 기억해야 할 것이다.

01. 한 번에 한 가지만 하는 짧은 코드를 만든다.

; 짧은 코드는 코드를 보는 사람의 이해를 돋는다.

02. 코드에 중복이 없다.

; 코드의 중복은 오류의 온상이다.

03. 정의된 역할 이상은 하지 않는다.

; 한가지 역할만 제대로 하는 짧은 코드를 짜는 것이 좋다.

04. 계층적인 구조와 균형을 유지하고 있다.

; 계층화 위반은 우연에 의해서 발생하지만, 충분히 개선할 수 있다. 코드는 각각의 디렉토리와 파일에 고르게 분포하는 것이 좋다.

05. 한 가지 변경 이유에 대해서 변경하는 부분은 한 곳이다.

; 잘 정의된 작은 변경을 위해서 코드의 여러 곳을 수정하고 있다면 문제가 발생할 가능성이 높다.

06. 기능을 확장하기 쉽다.

; 모든 코드는 변경된다. 기능의 확장도 변경의 일부이며, 이해하기 쉬운 코드를 짜야 이것도 가능하다. 또한, 기능을 확장하기 위해서 여러 곳을 수정해야 한다면 구조적으로 문제가 있을 가능성이 높다.

07. 자료 구조의 구체적인 구현이 외부에 감춰진다.

; 일단은 모든 자료구조의 내부 구현은 외부에서 알 수 없어야 한다. 무슨 필드를 어떻게 사용할지는 자료구조를 접근하는 제공되는 함수에 한정되어야 한다.

08. 인터페이스가 간결하고 충분하다.

; 복잡한 인터페이스는 잘못 사용할 가능성이 높다. 필요한 인터페이스가 없다면, 그 자체를 직접 사용하려고 할 것이다.

09. 모듈 들이 각각 재활용이 될 수 있다.

; 모듈화 되지 않은 코드는 재활용하기 위해서 큰 덩어리로 그냥 사용해야 한다. 모듈이라고 정의 하더라도 독립적으로 사용 되지 못한다면 모듈이 아니다.

10. 자동화된 단위 테스트가 존재한다.

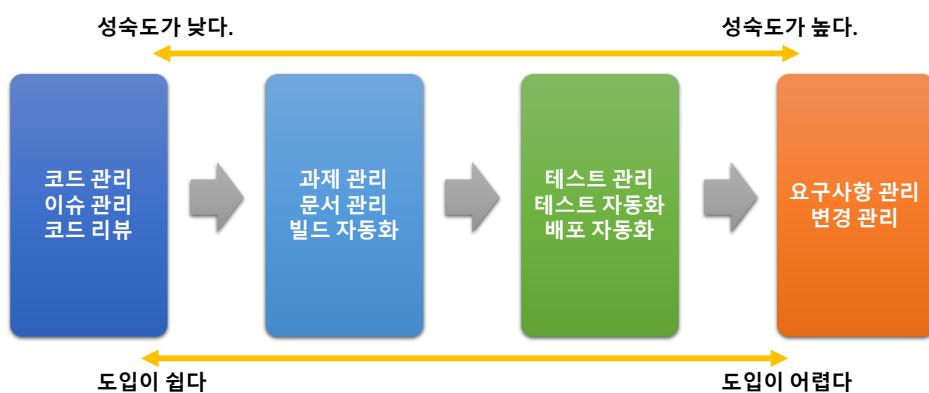
; 코드의 변경 후 자동화된 검증이 없다면 불안감에 수정조차 못한다.

이상과 같이 10가지를 나열해 보았지만, 순서가 중요한 것은 아니다. 물론, 여기에 추가해서 더 구체적인 것들도 있을 것이다. 스스로가 중요하다고 생각되는 요소들을 충분히 나열해보고, 그런 것들을 자신의 코드를 검사하는 체크리스트로 활용하는 것도 좋은 방법이다. 완결된 코드(완벽한 코드)는 없으며, 모든 코드는 완전한 상태로 도달하는 과정에 있다는 것이 정확할 것이다. 따라서, 코드는 지속적인 개선과정을 통해서 더 좋은 코드로 변화될 수 있다. 어느 곳에 비중을 두는가에 따라 달라지겠지만, 이런 규칙들이 개발자의 생각을 구속 하지는 않는다. 오히려, 그 생각을 더 명확하게 구현할 수 있도록 도와주며, 버그가 발생할 수 있는 복잡함을 사전에 제거할 수 있도록 도와줄 것이다.

[개발 툴(Tool)에 대해서]

소프트웨어 개발에서 툴은 절대적으로 필요하다. 툴이 생산성을 획기적으로 높이지는 않지만, 그렇다고 전혀 도움이 안되는 것은 아니다. 예를 들어, 명령라인에서 편집기를 열어 코딩하는 것보다, 통합 개발환경(IDE:Integrated Development Environment)을 사용하는 것이 편리하다. 코딩과 디버깅, 실행을 한번에 할 수 있기 때문이다. 그리고, 다양한 편집 기능을 이용해서, 코드를 이곳 저곳 뒤지지 않더라도 코드의 다양한 정보를 얻을 수 있다.

물론, 너무 많은 정보를 제공해서 문제가 되기도 한다. 예를 들어, 구조체의 내부 구조를 직접 접근해서 조작하는 코드를 작성하는 경우가 종종 있는데, 직접적인 자료구조의 내부조작은 자료구조가 정의된 모듈 내로 한정하는 것을 위반하는 사례다. 코딩 할 때 그런 내부적인 정보에 의존하는 것은 변경에 취약한 코드를 만들기 쉽다. 따라서, 툴을 사용하더라도 원칙을 지키고 유지하는 것은 도움이 된다. 그리고, 툴이 주는 편리함을 이용해서 가능한 자신의 코드를 다듬을 수 있으면 하면 된다.



개발 환경은 개발자에게 직접적인 도움을 주지만, 그 외에도 도움을 받을 수 있는 다양한 툴 들이 존재한다. 주로 상업적인 툴 들이 많지만, 공짜로 이용할 수 있는 오픈소스 툴 들을 사용하는 것도 좋다. 여기서 이야기하는 툴 들이 전부는 아니며, 자신에게 적합한 툴을 직접 모아서 항상 정리해 두면 좋을 것이다. 좋은 목수는 연장을 가리지 않는다고 하지만, 좋은 개발자는 적합한 연장을 시기 적절하게 정말 잘쓰는

사람이다.

01. 실행하지 않고 코드의 오류를 찾아주는 정적 분석기

; CodeSonar와 Coverity 같은 것이 대표적인 상업용 툴이다. 각 종 룰에 기초해서 문제를 일으킬 가능성이 있는 코드가 어디에 있으며, 왜 문제가 되는지, 해결할 수 있는 방법은 무엇인지 등을 알려준다. 그 외에도 각종 측정된 결과값(Metric들)을 보여주기에, 코드를 분석하기 쉽게 만든다. 예를 들어, 계층화를 위반한 경우나 지나치게 비대한(Fat) 부분들을 찾을 수 있도록 해준다. 오픈 소스로 사용할 수 있는 CppCheck는 개발자가 간단히 사용해 볼 수 있는 툴이다.

02. 코드를 분석해서 문서화를 해주는 툴

; Doxygen이 대표적인 툴이다. 같이 사용되는 플러그인(Plugin)으로 Graphviz와 같은 것이 있으며, 코드의 호출 관계를 시각적으로 표현해 준다. 또한, 코드와 연계해서 전체 코드를 직접 돌아다니면서 볼 수 있도록 HTML문서로 만들어준다. 대부분의 경우 코드에 대한 자동문서화를 위해서 사용하며, 특정 주석의 형태로 코드 내에 문서를 위한 주석을 삽입하게 된다. Doxygen을 이용하면 분석 결과를 XML과 같은 형태의 요약으로 뽑아낼 수 있으며, 이를 통해서 소프트웨어의 품질을 측정할 수도 있다. 예를 들어, 각 함수의 파라미터 개수를 조사하는 것도 가능하다.

03. 복사된 코드가 얼마나 있는지를 알려주는 툴

; CPD(Copy & Paste Detector)가 대표적인 툴이며, 코드를 분석해서 얼마나 많은 복제된 코드들이 있는지 알려준다. 기준을 어떻게 설정하는가가 문제지만, 대부분 10라인(어떤 툴에서는 6라인) 이상이 동일하다면 복제된 코드로 봐야할 것이다. "DRY(Do not Repeat Yourself)"와 같은 소프트웨어 개발의 원칙을 코드 직접 적용해서 검증할 수 있는 방법을 제공한다. 나중에 혹시 코드의 크기가 큰 경우에도 이를 활용해서 복제된 코드를 함수화 시키는 방법으로 코드 사이즈를 줄이는 일도 할 수 있다.

04. 함수의 길이를 측정하는 툴

; CodeSonar나 Coverity와 같은 툴은 소스 코드를 분석해서 각종 정보를 알려준다. 결과의 일부로 함수의 길이를 측정한 값도 알 수 있다. 각각의 함수의 길이를 어떻게 유지해야 할지는 결정해야 하겠지만, 30~100라인(코딩 룰에 따라 24~200라인까지) 정도의 크기를 가지도록 함수를 짜야할 것이다. 물론, 함수의 길이가 더 작으면 작을수록 이해하기는 쉬워지는 경향이 있으며, 함수의 파라미터 갯수도 줄어드는 경향이 있다. 개발자들은 함수의 길이에 대해서 신경을 많이 쓰지 않는 편이지만, 이는 심각한 문제가 될 수 있다. 될 수 있으면 짧은 함수를 많이 만들어서 문제를 해결하는 방향이 좋다.

05. 코드를 리뷰하기 위해서 사용하는 툴

; ReviewBoard와 같은 오픈소스 툴은 사용하기는 조금 어렵지만 웹에서 코드에 대한 리뷰를 할 수 있고, 코드의 비교와 리뷰 결과를 로그로 볼 수 있다는 점에서 괜찮은 툴이라고 생각한다. 코드 리뷰를 오프라인에서 하는 것은 상당히 많은 비용이 들어가는 행위로(예로, 5명이 1시간을 회의하면 대략 250,000원이 들어간다고 이야기 한다. 아마도 더 비쌀 것이다.) 중요한 코드가 아닌 상시적인 리뷰에 활용해 볼 수 있을 것이다. 코드 리뷰는 당연히 안하는 것보다 하는 것이 좋고, 상시적으로 하는게 그나마 변경을 줄이는 길이다. 상업용 툴로는 CodeCollaborator나 Crucible 등이 있다. 참고로 코드 리뷰는 가장 많은 버그를 찾아내는 방법이다.

06. 코드를 통합해서 빌드(Build)하는 툴

; 기본적으로 "Make" 스크립트는 알고 있어야 한다. 특히, 신입 개발자의 경우에는 "make" 명령어의 사용법과 "Makefile"을 작성하는 방법을 잘 알고 있어야 한다. 보통의 경우 빌드를 담당하는 개발자가 제일 개발을 잘하는 경우(혹은, 개발에 능한 개발자가 빌드를 담당)를 많이 볼 수 있다. 그리고, 나중에 빌드 스크립트를 수정하기 위해서도 반드시 이해해야 한다. 주로 명령줄에서 빌드하는 경우에는 "make" 명령어를 직접적으로 사용하겠지만, IDE와 같은 경우에는 내부적으로 진행되는 과정이라 제대로 눈에 보이지 않는다. 최근에는 "Cmake"와 같은 툴을 주로 사용하는 경향이 있으며, 플랫폼이나 OS에 상관없이 빌드할 수 있도록 만들 때 주로 이용한다. 자동으로 통합 빌드를 실행시키기 위해서는

Bamboo나 Jenkins도 활용해 볼 수 있을 것이다.

07. 코딩 룰을 검사하는 툴

; 코딩 스타일은 각종 통합 개발환경에서 일괄적으로 맞출 수 있는 방법을 기본적으로 제공하는 경우가 많다. 하지만, 특정 코딩 표준에 맞는 코드를 작성하는지를 검사하는 도구는 구매해서 사용해야 한다. 예를 들어, "CodeInspector"와 같은 툴은 MISRA-C 2012이외에 다양한 코딩 표준을 제공하고 있으며, 검사 결과 및 위반 이유와 수정방향에 대한 구체적인 정보를 알려준다. 물론, 이런 툴들이 잘못된 정보를 주는 경우도 있다(False Alarm or False Positive). 하지만, 그렇더라도 자신의 코드가 정말 문제가 없는지는 한번 더 검토해 봄야 한다. 참고로 CodeSonar나 Coverity, QAC/C++도 사용할 수 있다.

08. 코드를 실행해 보기 위한 툴

; 만약, 시스템에 치명적인 영향을 줄 가능성이 있는 프로그램을 개발한다면, 가상적인 실행 환경이 필요하다. 즉, 시뮬레이터나 가상 머신등이 이를 위해서 사용할 수 있다. 실제 시스템 상에서 실행하게 되면, 문제가 발생했을 때 복구하고 다시 실행하는데 시간이 오래 걸리기 때문이다. 가상 머신은 이미 오픈소스로 나온 것들이 있으니 그것들(VirtualBox)을 이용하면 될 것이다. 시뮬레이터의 경우에는 사실상 자신이 만들려는 제품에 맞는 것들이 없기에, 직접 만들어야 할 때도 있을 것이다. 물론, 아주 정밀할 필요까지는 없지만, 최소한 원하는 기능들은 제공되어야 한다. 테스트 프레임워크도 일종의 가상적인 테스트 환경을 구축하는데 사용할 수 있다.

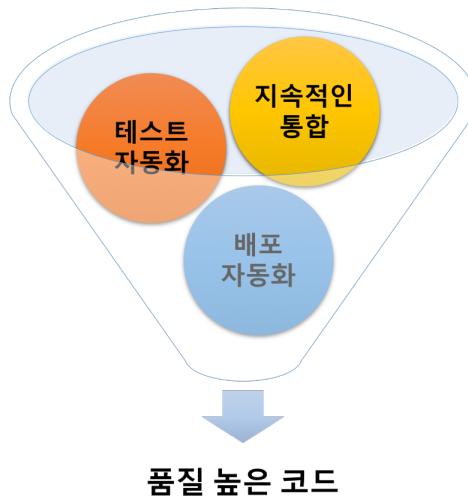
09. 저장소를 관리하기 위한 툴

; 오픈 소스로 Subversion과 같은 툴이 대표적이다. 혼자서 개발하더라도 이런 툴들은 반드시 사용해야 한다. 예전에는 ClearCase와 같은 툴들이 많이 사용되었지만, 최근의 경향은 클라이언트(즉, 개발자 PC)측을 가볍게 만들어주는 것을 선호한다. Perforce가 대표적인 경우다. Perforce의 경우에는 "스트림(Stream)"이라는 개념이 있어서, 개발되는 코드를 계층적으로 관리할 수 있도록 해준다. 예를 들어, "Mainline"->"Development"->"Release"와 같은 스트림들을 만들어서, 통합(Merge)과 가지(Branch)를 만들 수 있다. 이런 툴들에서 사용하는 용어는 조금씩 차이가 있을지는 모르지만, 대부분 비슷한 개념들을 사용하기에, 한 툴을 잘 익히면 다른 툴을 사용하는데도 큰 지장은 없다. 분산된 개발환경을 위해서는 Bazaar, GitHub도 최근에는 많이 사용되고 있다.

10. 과제를 관리하기 위한 툴

; 과제를 관리하기 위해서는 문서화와 이슈 추적을 동반한다. 대표적인 오픈 소스 툴로는 "Trac"이 있으며, 문서화를 위한 위키(Wiki)형식의 웹페이지 작성과 이슈 추적을 위한 티켓(Ticket) 발행이 가능하다. 각종 플러그인들도 많이 제공되기에, 확장 기능들도 이용하기 쉽다. 코드의 전반적인 품질 수준을 알기 위해서는 "SonarQube"라는 툴도 활용할 수 있다. 코드를 개선하기 위해서 필요한 노력(Effort)의 양을 기술적인 빚(Technical Debt) 형태로 알려주기 때문에, 이를 보고 상대적인 코드의 품질 수준을 추측해 볼 수도 있다. 다양한 상업적인 혹은 오픈 소스 플러그인들이 제공되기에, 품질과 관련된 해서 활용해 볼 수 있다. Mantis나 Bugzilla와 같은 툴은 공짜로 사용할 수 있으며, TeamForge와 같은 툴은 Subversion과 통합해서 사용하기 쉬울 것이다. 비용이 조금 들기는 하지만, Confluence와 JIRA를 조합해서 사용하는 것도 고려해 볼만하다. 특히, Confluence와 JIRA를 조합해서 스크럼(SCRUM)과 같은 애자일(Agile) 방법론을 적용하기 쉽게 만들 수 있다.

이상에서 10가지의 필요한 툴들을 나열해 보았지만, 이외에도 과제마다 혹은 분야마다 다양한 툴을 활용해서 개발자의 부담을 줄여주려는 노력을 할 수 있을 것이다. 좋은 툴을 사용하는 것은 개발자가 더 가치있는 일에 집중할 수 있도록 만들기 위함이지, 툴이 사람을 대체하는 것이 아니다. 도구는 도구일 뿐이라는 말이다. 좋은 목수가 되기 위해서는 좋은 도구가 무엇인지도 알아야 한다. 그리고, 그 도구를 최고로 다룰 수 있는 사람은 다른 도구로 바꾸더라도 불평하지 않을 것이다(하지만, 도구 자체가 없는 것은 불평 요인이 될 수 있다.).

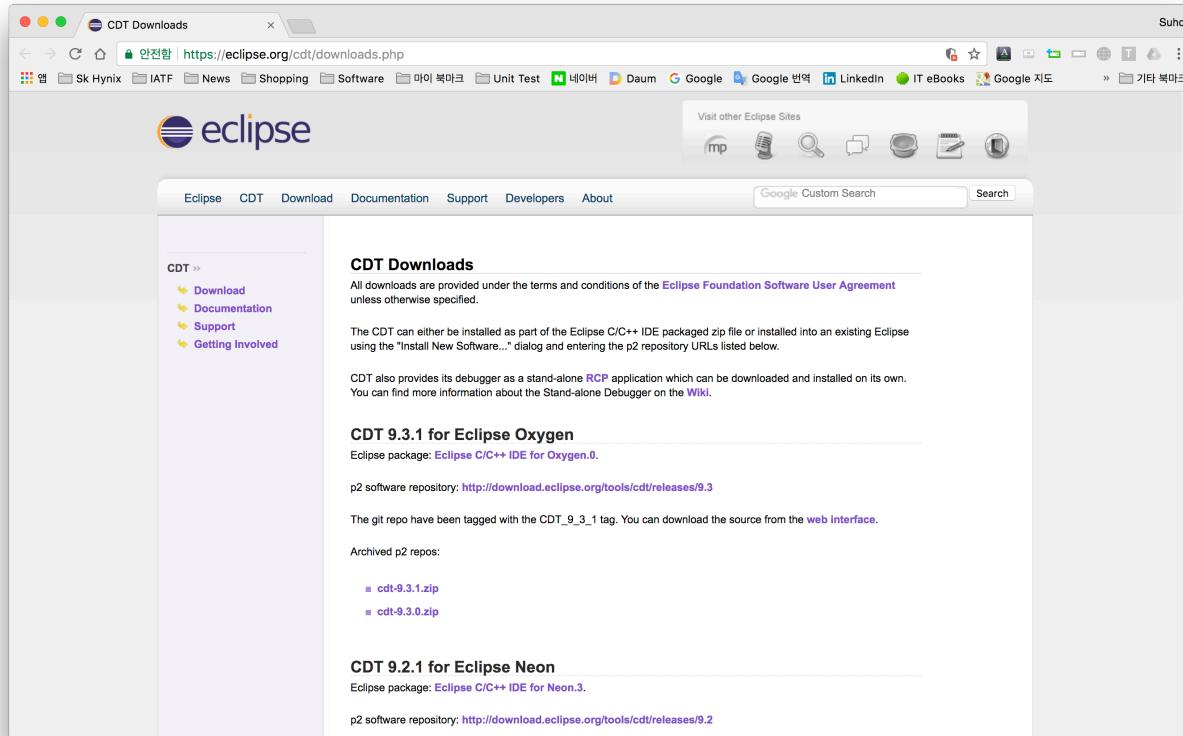


비용이 든다고 말 할지는 모르지만, 도구에 들어가는 비용보다 더 많은 가치를 만들 수 있다면 충분히 투자해 볼 만한 가치가 있다. 예를 들어 Coverity와 같은 상업적인 툴은 비싸다. 하지만, 그런 도구를 통해서 치명적인 버그 1개를 줄일 수 있다면, 당연히 사용하는 것이 더 나은 선택이다. 최악의 상황은 아주 작은 가능성에 불씨가 남을 때 발생한다. 아무리 사고 확률이 낮다고 이야기 할지라도, 아무도 예상할 수 없는 사소한 것에서 문제가 나올 수 있다. 우리가 할 수 있는 최선의 선택은 가능한 그런 것들을 최대한 막으려고 노력하는 길밖에 없다. 도구는 사람을 돋기 위한 것이지 사람을 괴롭히기 위한 것이 아니며, 또한 그 도구를 어떻게 사용 하는가는 항상 사람에게 달려있다.

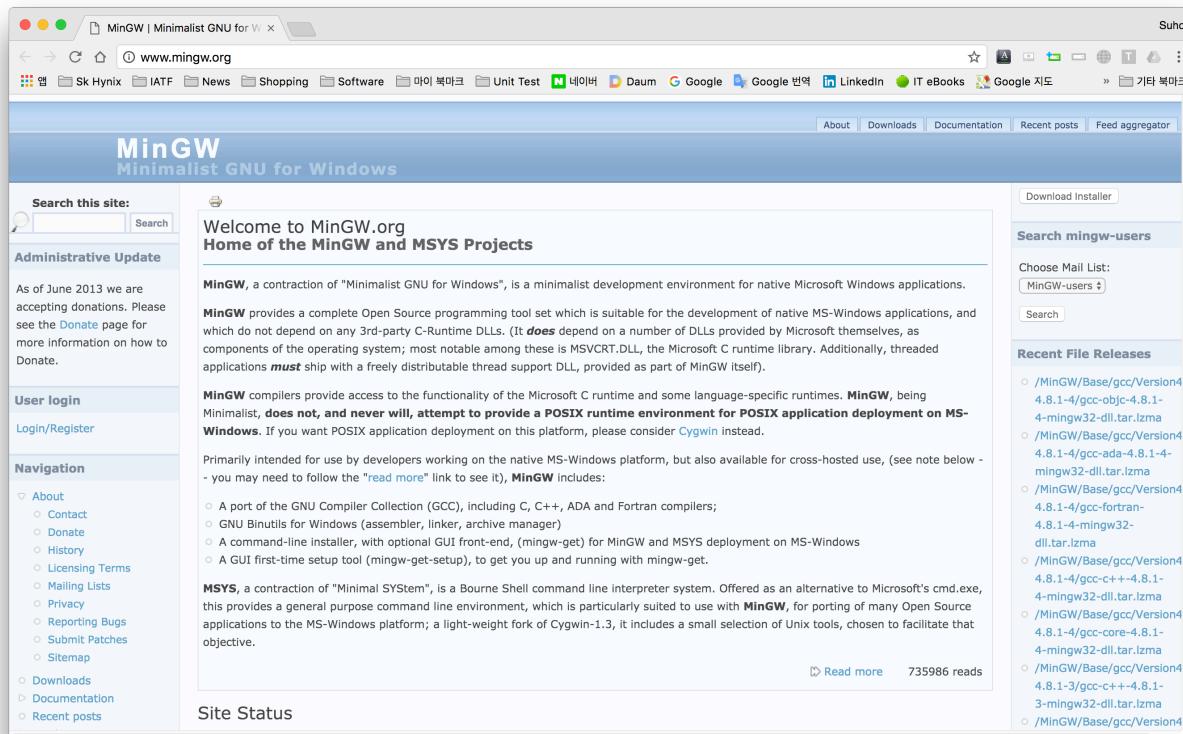
[개발환경 익히기]

개인용 PC가 일반화되기 시작한 90년대에는 아직 인터넷이 보급되기 전이라, 필요한 프로그램들을 어둠의 경로를 통해서 받거나, 혹은 시장에서 사야했다. 현재는 공짜로 사용할 수 있는 툴(Tool)들이 많이 있으며, 그중에 대표적인 Eclipse를 기반으로 프로그램 개발에 대해서 설명하겠다. Eclipse는 원래 Java 프로그램을 개발하기 위한 통합개발환경을 목표로 오픈소스(Open Source) 프로젝트로 진행되었지만, 현재는 다양한 언어로된 코드를 작성해서 실행시켜 볼 수 있다. 또한, Plugin 형태로 설치 가능한 많은 확장 프로그램들도 찾을 수 있다(일종의 마켓도 가지고 있다.).

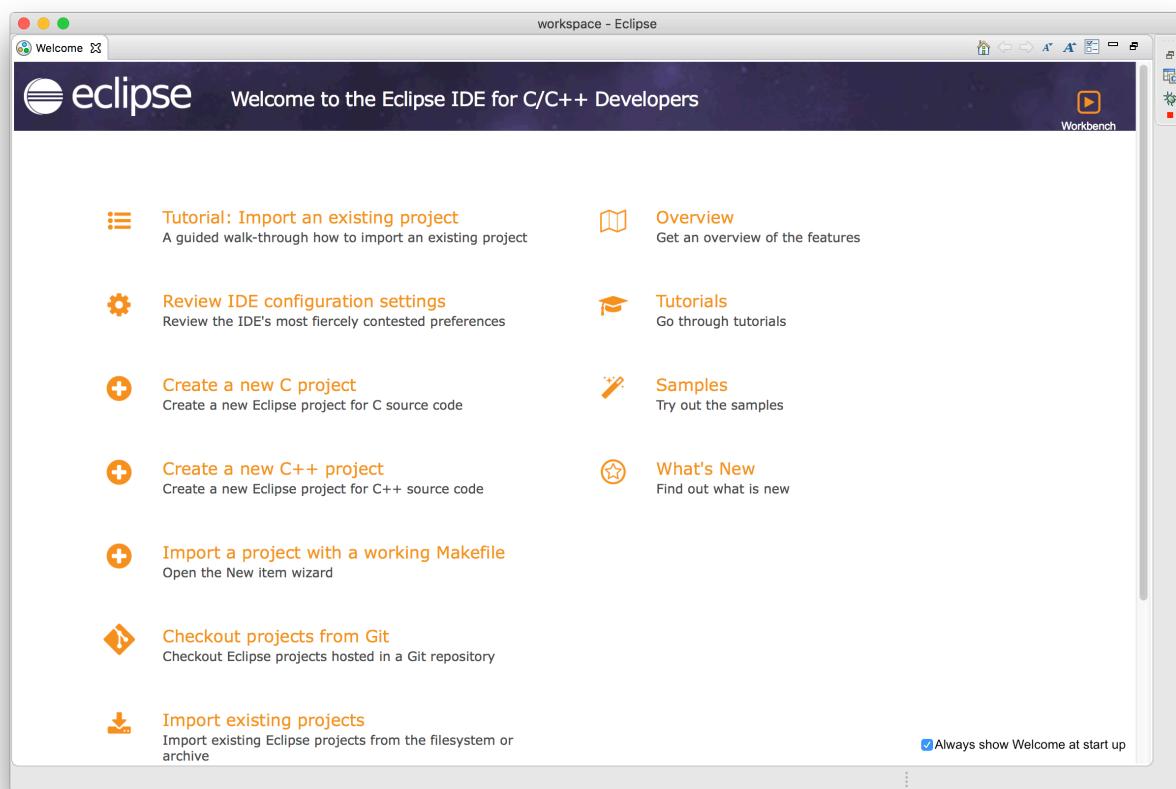
C언어 개발환경을 위해서는 Eclipse CDT(C/C++ Development Tooling)가 필요하다 이름에서 알 수 있듯이 C와 C++ 두개의 언어를 사용할 수 있는 IDE(Integrated Development Environment)를 제공한다. 다운로드는 "<http://www.eclipse.org/download/>"에서 받을 수 있다. 개발환경이 설치되는 운영체제를 윈도우즈 환경이라고 가정하도록 하겠다.



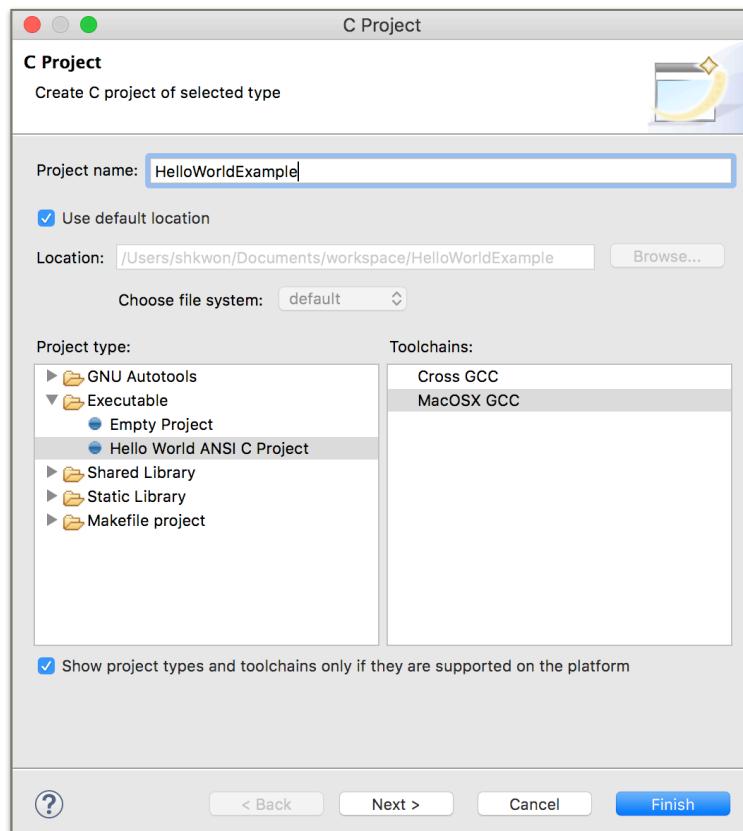
설치는 특별한 과정이 필요하지 않으며, 그냥 압축된 파일을 해제해서 원하는 위치에 두면 된다. 추가적으로 C/C++ Compiler를 필요로 하는데, 윈도우즈 환경에서는 “MinGW 32/64”를 설치해서 사용할 수 있다. 다운로드는 [“http://www.mingw.org”](http://www.mingw.org)에서 받을 수 있다.



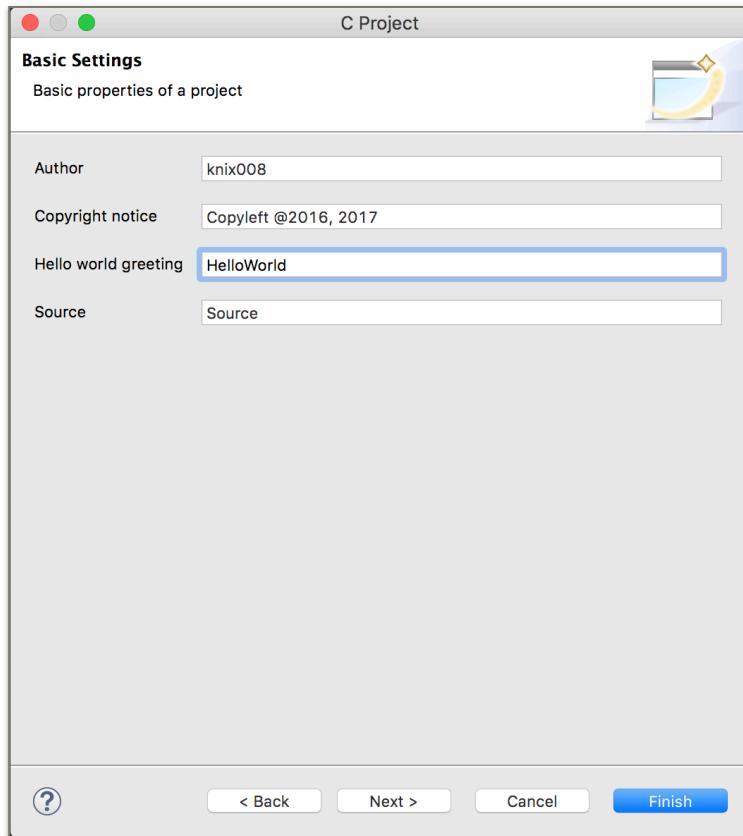
MinGW는 Installer를 받아서 윈도우 환경에서 설치하면 될 것이다. 다운로드 받은 Eclipse의 압축을 해제한 후에, 압축이 해제된 디렉토리에서 eclipse를 실행시켜보자.



하나의 C언어 프로젝트를 만들어 보도록 하겠다. "File --> New --> C Project"를 선택한다. 여기서는 이미 Eclipse에 포함된 C언어 프로젝트를 위한 설정을 사용하도록 한다.

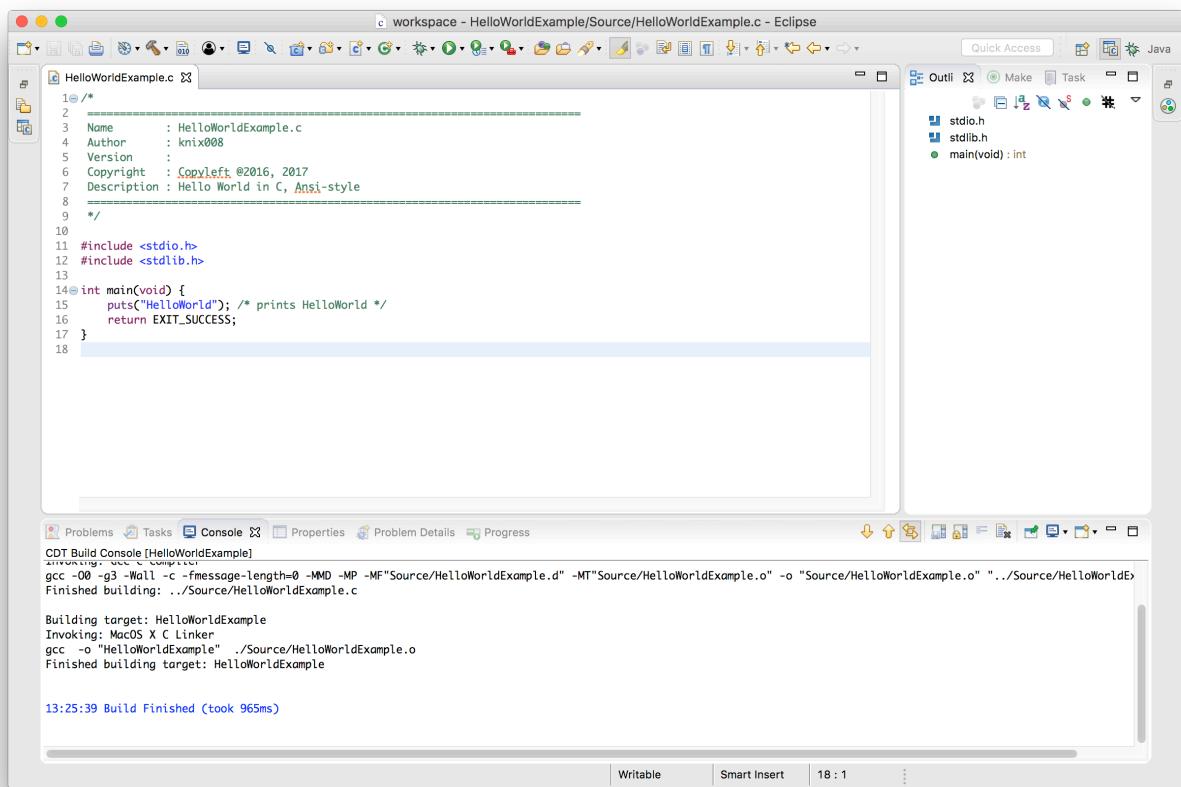


"Project name"에는 "HelloWorld"를 적어주고, "Use default location"을 체크한다. "Project Type"은 "Executable--> Hello World ANSI C Project"를 선택하고, "Toolchains"는 "MinGW GCC"를 선택하도록 한다. 마지막으로 "Next"버튼을 눌러서 다음으로 진행한다.



"Author"에는 자신의 이름을 넣어주면 된다. 나머지는 그대로 두고, "Next"를 선택하도록 한다. 물론, "Finish"버튼을 눌러도 상관없지만, 여기서는 마지막까지 선택하도록 하겠다. "Next"를 선택했다면 컴파일을 할 수 있는 두가지를 선택해야 한다. 각각은 "Debug"와 "Release"이다.

실행 파일은 기본적으로 "Debug" 모드와 "Release"모드 두 가지를 설정할 수 있다. 여기서는 기본으로 설정된 것을 그대로 두고, 마지막 "Finish"버튼을 누르도록 한다. 디버그 모드는 컴파일할 때 "-g(디버그)"옵션을 활성화 시키며, 배포 모드에서는 "-O2(최적화)" 옵션이 활성화 된다.



HelloWorldExample01 프로젝트의 Eclipse 개발환경 화면 예에서 보듯이, 간단한 C 프로그램 파일이 생성되었다. 잠시 내용을 살펴보면 먼저 프로젝트 생성단계에서 입력했던 것들이 가장 위쪽에 주석으로 처리되어 나올 것이다. 코드는 아래와 같다.

```

#include <stdio.h>
#include <stdlib.h>

int main(void) {
    puts("!!!Hello World!!!"); /* prints !!!Hello World!!! */
    return EXIT_SUCCESS;
}

```

두 개의 "#include"는 출력 함수인 "puts()"과 "EXIT_SUCCESS"를 사용하기 위해서 필요하다. "main()" 함수는 C언어에서 프로그램의 시작점을 나타내며, 입력으로는 아무것도 받지 않는다(void). "main()" 함수의 출력은 "int(Integer)" 값이 호출 복귀 값(Return Value)으로 사용된다. "main()" 함수가 하는 일은 간단히 "!!!Hello World!!!"를 출력하고, "EXIT_SUCCESS"를 호출의 복귀 값을 사용한다.

Eclipse의 화면에서 좌측은 "Project Explorer"를 보여주며, 우측은 선택된 소스코드 파일의 중요한 자료구조를 보여준다. 이것을 이용해서 필요한 코드를 편집하고, 원하는 것들을 찾을 수 있다. 다른 다양한 기능들도 있으니, 한번씩 눌러서 실행해 보기 바란다(이 책은 Eclipse를 위한 것이 아니기에, C언어를 익히는데 필요한 것만 볼 것이다.).

[“Hello, World!!!”를 다시 보다.]

프로그램을 배우는 가장 쉬운 방법은 책에 등장 하는 모든 코드를 직접 만들어서 실행해 보는 것이다. "Hello, World!!!"는 C나 C++언어를 보면 가장 먼저 제시되는 예제 프로그램이다. "Main"으로 시작해서 간단히 "Hello, World!!!"라는 문장을 화면에 출력한다. 아래의 코드를 보도록 하자. 일단 아래의 코드를 HelloWorld.c라는 파일에 저장했다고 가정하자.

```
#include<stdio.h>

int main(int argc, char** argv) {
    printf("Hello, World!!!\n");
    return 0;
}
```

간단한 코드다. 하지만, 초보자에게는 대단히 어려울 수 있는 코드다. 왜냐면 모르는 언어를 처음 배울 때, 단어의 뜻과 문법에 압도될 수 있기 때문이다. 이 코드는 단순히 C코드가 어떻게 동작하는지를 알려주기 위해서 작성된 것이지만, 처음보는 사용자는 이해하기 어려울 수 있다는 사실을 간과 하고 있다. 그리고, 모든 일반적인 언어(일본어, 중국어, 영어 등등)를 배울 때 가장 쉬운 방법은 그냥 들은 것을 따라 해서 반복하는 것이라는 점을 내포하고 있다.

이해하기 쉬운 코드를 짜야하는 것과 더불어 습관적으로 좋은 코드를 짜는 방법을 반복해서 익혀야 한다. 먼저, 이 코드에서 개선할 점부터 찾도록 하자. 개선할 점은 파일의 이름에서부터 출발 한다. "HelloWorld.c"라는 파일 이름 보다는 내부의 코드가 무슨 일을 하는지 정확히 알려주는 이름을 사용하는 것이 좋다. "PrintHelloWorld.c"는 어떨까? 일단 파일을 열지 않아도 무슨 일을 하는지는 알 수 있다. 급한 경우에는 파일 이름만 보고도 쉽게 원하는 일을 하는 코드를 찾을 수 있고, 파일 검색 등도 이용 할 수 있다. 두 번째는 "main()"함수가 무엇인지를 알려주는 것이 좋다(물론, 이 경우는 아무것도 모르는 신참 개발자는 대상으로 한다.). "main()"함수 위에 간단히 /* Start Point */와 같은 Comment를 적어 준다면, 코드의 시작점이 어디인지 충분히 알려줄 수 있을 것이다.(나중에 보겠지만, 주석이 많다고 좋은 코드는 아니다.)

"printf()"함수는 시스템 호출(System Call)을 동반한다. 그리고, "main()"함수의 수준에서 호출하기에는 저 수준(구체적인) 함수에 속한다. 즉, 추상화 수준이 "main()"함수보다 한 단계만 아래에 있다고 볼 수 없다. 따라서, "int printHelloWorld()"라는 함수로 분리시켜준다. 이때는 "main()"함수가 하는 일이 저 수준 함수의 호출보다는 더 효과적으로 이해가 될 것이다. 그리고, "return"문은 "printHelloWorld()" 함수의 호출을 돌려주는(Return)하는 것을 바꿀 수 있다. 따라서, 다시 적은 코드는 아래와 같다.

```
#include<stdio.h>

int printHelloWorld(void) {
    printf("Hello, World!!!\n");
    return 0;
}

/* Start Point */
int main(int argc, char** argv) {
    return printHelloWorld();
}
```

조금 개선되었다고 보이나? 사실 초보자가 이런 모든 과정을 볼 이유는 없다. 여기서 단지 몇 가지를 이야기 하고 싶었기 때문에 적어본 것이다. 첫 번째는 코드로 이해가 어려울 경우에만 코멘트로 설명해야 한다는 것과. 두 번째는 함수의 추상화 수준은 자신이 하는 일을 한번에 설명할 수 있는 이름을 가져야하

며, 함수의 내부 구현 코드들은 함수의 이름보다 한 단계 낮은 추상화 수준을 가져야 한다는 점이다. 세 번째는 함수의 이름은 이해하기 쉬워야 하며, 하나의 함수는 한가지 일만 해야 한다(이것을 Single Responsibility Principle"이라고 한다.). 짧은 함수가 이해하기 쉽우며, 모든 함수는 복귀값(Return Value)을 가지는 것이 좋다(최적화에는 좋지 않을 수 있지만)는 것, 인자(Parameter)가 없는 함수가 최고라는 것이다.

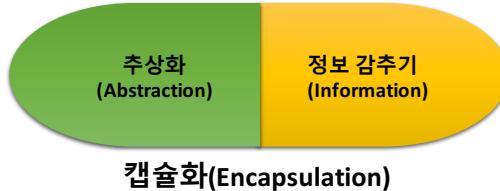
"Hello, World!!!" 프로그램을 보면, main()함수와 printHelloWorld()함수간의 논리적인 일관성이 부족하다. 즉, main()함수가 하는 일과 printHelloWorld()가 하는 일은 호출관계가 있지만, main()함수와 같은 파일에 있기에 파일 이름(printHelloWorld.c)과 어울리지 않는다. 따라서, printHelloWorld()함수를 main.c(혹은, main()함수를 분리해서 다른 파일로)에서 분리해서 새로운 파일로 만드는 것이 옳다. 분리된 파일을 "printHelloWorld.c"라고 하자. 그리고, main()함수가 정의된 파일은 "c_main.c"라고 하자. c_main.c파일에 남은 main()함수는 아래와 같다.

```
#include<stdio.h>
```

```
/* Start Point */
int main(int argc, char** argv) {
    return printHelloWorld();
}
```

이전보다 좀 더 깔끔하게 정리된 것을 볼 수 있다. 즉, "c_main.c"파일에는 "main()"함수 이외에는 정의가 없는 것을 볼 수 있다. 대부분의 소프트웨어 개발자들은 관례적으로 "c_main.c" 파일이 프로그램의 시작을 정의한 파일이라는 것을 인식할 것이고, 잠시 훑어본 후에는 다시 열어보는 일이 없을 것이다. 한 가지 빠진 부분은 "printHelloWorld()"함수를 "main()"함수가 찾을 수 없다는 점이다. 즉, 어디에도 정의가 나오지 않는다. 따라서, "printHelloWorld()"함수를 찾을 수 있도록 도움을 주어야 한다.

먼저 해야 할 일은 선언과 구현을 분리하는 일이다. 즉, 인터페이스(Interface)와 구현(Implementation)을 각각 다른 파일로 만들어서, 구현을 인터페이스 너머로 감춰주는 일이다. 이렇게 하는 이유는 인터페이스의 일관성을 유지하면서, 기능을 확장하거나 변경을 쉽게 할 수 있기 때문이다. 즉, 구체적인 것에 의지하면 변경이 어려워지거나, 변경의 영향으로 버그가 유발될 가능성이 높기 때문이다. 위의 경우에는 헤더 파일을 통해서 인터페이스와 구현이 분리될 수 있다.



선언과 정의를 분리하는 것은 중요한 개념이다. 객체지향 프로그램 언어에서 제공하는 캡슐화(Encapsulation)를 이용한 정보은닉(Information Hiding)을 구현하는 이유도 동일하다. 즉, 구체적인 것을 감춤으로 해서 변경의 자유도를 높이는 것이다. 변경의 자유도가 높다는 말은 호출하는 코드와 호출되는 코드간에 연결고리가 약해져서, 서로 상대방을 알지 못하는 상황에서 인터페이스만 일치시키면 변경이 자유롭다는 것이다. 물론, 인터페이스가 제공해야 할 기능은 반드시 호출되는 코드에서 제공되어야 한다. 그렇지 않다면, 인터페이스라는 계약(Contract)을 위반한 것이기 때문이다.

헤더 파일이 하는 역할은 간단하다. 즉, 각종 필요한 선언(Declaration)들을 넣어서 필요한 모듈(Module)에서 참조하도록 만든는데 있다. 어떤 자료형이나 함수의 선언들을 미리 가지고 있어서, 그것을 필요한 부분에서 선언들을 참고해서 코딩할 수 있도록 도움을 주는 것이다. 따라서, 헤더 파일에는 가

능한 최소한의 정보가 포함되어야 한다. 많은 정보가 있을수록 좋다고 생각할 수도 있겠지만, 필요한 만큼의 정보를 외부에 제공하는 것이 핵심이다. 그리고, 정의는 가능한 구현 파일에 숨기고, 인터페이스만 외부에 제공되어야 한다. 핵심 자료구조와 같은 것들도 될 수 있으면 외부에 제공되지 않아야 한다. 단순한 형(Type)정도의 정보만 제공되면 충분한다. 여기서는 "printHelloWorld()"함수를 위한 헤더 파일을 보도록 하자. 파일의 이름은 "printHelloWorld.h"라고 하자.

"printHelloWorld()"함수는 내부에서 "printf()"함수를 호출하고 있다. 만약 "printf()"함수를 사용해서 화면(Console)상에 글자를 쓰는 것이 아니라, 파일로 로그(Log)를 저장하는 목적이라고 한다면 어떨까? 즉, 단순히 "printf()"함수를 호출하는 것 보다는 사용자의 설정에 맞춰서 출력 방향을 변경해 줄 수 있어야 한다. 이를 위해서는 "printf()"함수 자체에 대한 추상화도 고려하게 되는데, 이때 사용할 수 있는 방법도 역시 인터페이스를 정의하는 것이다.

```
void (*PRINT)(const char* message);
```

이렇게 형식을 정하고, 화면으로의 출력과 로그의 출력에 대해서 각각 "printMessageOnConsole()", "printLogMessageOnFile()"과 같은 함수를 같은 형식으로 선언하면 된다. 각각의 함수는 화면에 대한 출력과 파일에 대한 출력으로 구분해서 구현한다. 그리고, 각각의 함수는 "print"라는 범주에 속하기 때문에 같은 파일내에 구현될 수 있다. 나중에 더 기능이 세분화 되면, 이것도 분리된 파일로 나누어 주면 된다.

만들어진 "c_main.c" 파일과 "printHelloWorld.c" 파일은 각각 분리되어 재사용될 수 있다. "c_main.c" 파일은 테스트 용도로 재활용이 가능하고, "printHelloWorld.c" 파일은 다른 프로그램에서 파일 자체를 이용할 수 있다. 하지만, 여기에도 추가할 부분이 존재한다. 즉, C와 C++ 코드를 같이 사용하기 위해서는 C 파일에 추가적으로 해줘야 할 일이 있다. 즉, C로 정의된 함수를 C++에서 찾을 수 있도록 만들어주는 것이다. 이를 위해서는 "extern "C""를 사용한다. "extern"은 외부에서 정의된 것을 찾으라는 뜻이고, ""C""는 그것이 C언어로 표현된 것이라는 뜻이다. C++ 컴파일러는 이 문장을 만나면, 자신이 가지고 있는 기본적인 기능보다는 "C"언어의 규칙을 따르도록 코드를 생성한다.

```
#ifdef __cplusplus
extern "C" {
#endif

int printHelloWorld(void);

#ifndef __cplusplus
}
#endif
```

위와 같이 해주면, "printHelloWorld()"함수를 이제는 C++에서도 활용할 수 있다. 따라서, C++에서 C 코드를 재사용하기 위해서는 헤더 파일에 위와 같이 정의해 주는 것이 필요하다. 수정하지 않았을 경우에는 C++ 컴파일러는 해당 함수를 찾을 수 없다는 링크(Link)오류를 발생시킬 것이다. 실무 개발에서는 C언어 및 C++언어를 섞어서 사용할 수 있으며, 이때는(C++언어 컴파일러를 사용하는 경우에는) 이런 과정이 반드시 필요하다.

한 가지 프로그램에서는 일반적으로 하나의 언어를 사용해서 구현하는 것이 일반적이지만, 이미 만들어진 라이브러리(Library)나 제 삼자(3rd Party)가 제공하는 바이너리(Binary)를 사용할 경우에는 개발 환경에 맞춰서 재사용할 코드를 변경해 주어야 한다. 이를 위해서 헤더 파일이나, 혹은 컴파일 옵션(Option)을 변경할 수 있도록 코드에 삽입하기도 한다(물론, 컴파일 옵션이 코드에 많이 추가되면, 코드를 읽는 것은 더 어려워질 수 있다).

[“Hello, World!!!”를 더 자세히 보기]

앞의 코드는 "Hello, World!!!"가 하드 코딩(hard coding)되어 있다. 따라서, 재활용의 측면에서 보자면 좋은 코드는 아니다. 좀 더 쓸모있게 만들기 위해서는 요청된 문자열을 출력하는 함수로 만드는 것이 좋을 것이다. 이를 위해서는 "Hello, World!!!"를 출력하는 함수와 문자열을 출력하는 함수로 다시 나눌 수 있을 것이다. 또한, 추가적으로 "Goodbye, World!!!"도 출력할 수 있도록 각각을 나누어진 함수로 구현할 수도 있을 것이다.

```
#include <stdio.h>
#include <stdlib.h>

void print_message(char *str) {
    printf("%s\n", str);
    return;
}

void print_hello_world(void) {
    print_message("Hello,World!!!");
    return;
}

void print_goodbye_world(void) {
    print_message("Goodbye, World!!!");
    return;
}

/* c_main.c */
/* Start Point */
int main(void) {
    print_hello_world();
    print_goodbye_world();
    return 1;
}
```

이전 코드에서 재사용 할 수 있는 부분을 새로운 API로 만들고, 직접적인 출력을 해야하는 일에서 분리시켰다. 이제는 좀 더 추상화가 되었다는 것과 나중에 새로운 메시지를 출력하더라도 변경되는 부분을 한정시킬 수 있다는 점이 달라졌다. 그리고, 해야할 일에 맞는 API를 만들어 기능을 추가했다는 것 ("print_hello_world()와 print_message()")이 변했다. 즉, 함수가 하는 일이 좀더 명확하게 구분 되었다는 점이 개선으로 볼 수 있다. 이렇게 하려는 목적에 맞는 함수를 정의하는 방법을 “인터페이스 세분화 혹은 분리(Interface Segregation)” 원칙이라고 부른다. 즉, 하려는 목적에 딱 맞는 인터페이스를 제공하는 것이다.

이와 같이 코드를 만들면 구체적인 것과 추상적인 것의 분리가 가능해지고, 나중에 코드를 재활용하기도 쉬워진다. 또한, 기능의 추가로 인해서 변경되는 부분도 최소화 될 가능성 높다. 나중에 이렇게 만들어진 코드들은 자신의 역할에 맞게 파일로 분리될 것이며, 코드들은 점점 더 모듈 및 서브 시스템으로 발전하게 될 것이다. 만약, 테스트가 추가 된다면, 테스트를 위한 분리도 필요할 것이며, 이럴 경우에도 좀 더 수월하게 코딩할 수 있을 것이다.

나빠진 부분은 코드가 좀 더 길어졌으며, 함수의 호출이 늘어났다는 점이다. 성능의 극한으로 끌어올리는 코딩을 하기 위해서는 이런 부분들을 조심해야 하겠지만, 사실 그런 코딩은 전체 코드에서 차지하는

비율이 적다. 일단은 구조적으로 코드를 만든 후에, 정말 필요한 부분의 성능을 측정해서 최적화를 시켜야 할 것이다. 추상화 수준에서 보자면, “main()”함수에는 하려는 일만 남았다. 각각의 호출되는 함수는 분리된 기능을 하나씩 대표한다고 볼 수 있으며, 공통적으로 사용하는 함수를 하위 계층으로 만들어서 호출하고 있는 것이다. 즉, 상위 사용자 인터페이스, 기능 로직 구현, 데이터 입출력과 같이 3단 구조로 만들어졌다.

```
/* print_message.h */
#ifndef __PRINT_MESSAGE__
#define __PRINT_MESSAGE__
void print_message(char *str);
#endif

/* print_message.c */
#include "print_message.h"
void print_message(char *str) {
    printf("%s\n", str)
    return;
}

/* print_helloworld_goodbyeworld.h */
#ifndef __PRINT_HELLOWORLD_GOODBYEWORDL_H__
#define __PRINT_HELLOWORLD_GOODBYEWORDL_H__

void print_hello_world(void);
void print_goodbye_world(void);
#endif

/* print_helloworld_goodbyeworld.c */
#include "print_message.h"
#include "print_helloworld_goodbyeworld.h"

void print_hello_world(void) {
    print_message("Hello,World!!!!");
    return;
}

void print_goodbye_world(void) {
    print_message("Goodbye, World!!!!");
    return;
}

/* c_main.c */
/* Start Point */
#include <stdio.h>
#include <stdlib.h>

#include "print_message.h"
#include "print_helloworld_goodbyeworld.h"

int main(void) {
    print_hello_world();
    print_goodbye_world();
```

```

    return 1;
}

```

위의 코드는 맡은 역할(Role)에 따라 파일을 나눈 것이다. 각각의 파일은 자신이 해야 할 일만 담당하며, 입출력에 대한 의존성을 가지는 부분도 다른 파일로 분리되었다. 호출되는 함수들을 위해서 각각의 구현 파일(".c")을 위한 헤더 파일을 가지고 있으며, "main.c"에서는 필요한 헤더 파일을 "#include"를 사용해서 넣었다. 이제 "main()" 함수의 내부는 추상화 수준이 맞는 것들로만 구성되었으며, 각각의 함수들도 자신이 해야 할 일이 명확히 하게 되었다. 파일 이름으로 무슨 일을 하는지도 충분히 짐작할 수 있다. 프로그램도 조금 더 읽기 쉽게 변했음을 알 수 있다.

물론, 지금까지 이야기 한 것은 "Hello, World!!!"를 출력하는 프로그램을 만들기 위해서는 과분하다. 사실 필요한 만큼만 구현하는 것이 최선이다. 여기서 이야기 하고자 하는 것은, 점차 요구되는 것들이 많아지면서 코드를 어떻게 구조화 시킬 수 있는가를 간략하게 보여주었을 뿐이다. 코딩이란 결국 변화에 대해서 능동적으로 대처해 나가는 과정의 연속이라는 것이다.

[생각의 추상화와 계층화]

소프트웨어를 개발하는 과정은 추상화된 해결책을 구체화시키는 반복 과정으로 이해될 수 있다. 이때 추상화되는 과정에서 필연적으로 계층구조를 가지는 모듈들로 만들어지며, 하나의 계층을 쌓아올리면 이전보다 한 단계 더 추상화된다. 추상화란 어떤 것을 묘사할 때, 세세한 부분에 대한 정밀한 그림을 그리는 것이 아니라, 대표할 수 있는 특징을 통해서 전체를 표현하는 것이다. 이런 것들은 다양한 부분에서 사용되고 있으며, 모델링도 그 일종이라고 할 수 있다. 사람이 다른 동물과 차이가 날 수 있는 부분이 이런 추상화를 통해서 구체적인 문제를 해결 할 수 있다는 점이며, 코딩에서는 계층화를 통해서 표현된다(특히, C언어와 같은 절차지향의 언어에서는).

```

#include <stdio.h>

int main( int argc, const char* argv[] ) {
    printf("Hello, World!!!\n");
    return 0;
}

```

위의 코드는 "Hello, World!!!"라는 문장을 화면상에 표시하는 간단한 프로그램이다. 파일의 이름을 "main.c"라고 두고, 이 파일이 위치한 디렉토리를 "HelloWorld"라고 만들면, 사람들은 그 디렉토리 이름만 보고도 대략적으로 무슨 일을 하는지 알 수 있다. 물론, "PrintHelloWorld"와 같이 더 묘사적인 이름을 사용하면, 그 내용을 보지 않고도 내부에서 무슨 일이 일어나는지를 알 수 있다. 만약, 파일 이름을 "print_hello_world.c"와 같이 고친다면 어떨까? 이제는 파일 이름만 보고도 무슨 일을 하는지 파악할 수 있다.

"main()" 함수는 모든 C언어로 만들어진 코드의 시작을 알려주는 함수이다. 하지만, 그 함수 내부에서 바로 호출되는 "printf()" 함수는 논리의 비약이 발생했다고 생각할 수 있다. 즉, "C코드의 시작함수 --> print()"는 관계가 어색하다. 즉, 추상적인 것이 급격하게 구체적인 것으로 변한 것이다. 이를 완화하기 위해서는 "print_hello_world()"와 같은 함수를 다시 정의할 수 있다.

```

#include <stdio.h>

static void print_hello_world() {
    printf("Hello, World!!!\n");
}

int main(int argc, const char* argv[]) {

```

```

print_hello_world();
return 0;
}

```

인터페이스 분리 법칙(Interface Segregation Principle)은 클라이언트의 요구에 각각 분리된 인터페이스(함수, 혹은 API : Application Interface)를 유지하는 것으로, 변화에 대응하는 한가지 방법으로 사용될 수 있다. 위의 코드는 단순히 "Hello, World!!!"라는 문자열을 출력하기 위해서 함수를 정의했다. 물론, 이런 것이 별로라는 것은 다 알 것이다. 따라서, 사람들은 좀 더 공용화된 인터페이스를 정의하고 싶어질 것이다. 따라서, 새로운 함수에서는 다양한 문자열을 입력으로 받을 수 있도록 만든다.

```

#include <stdio.h>

static void print_message(char *message) {
    printf("%s\n", message);
}

int main(int argc, const char* argv[]) {
    print_message("Hello, World!!!\n");
    return 0;
}

```

하지만, 이 코드는 단순히 화면상으로만 문자열을 보여줄 뿐이다. 우리는 다양한 장치로 문자열을 보여주고 싶다는 생각을 할 수 있다. 이를 위해서는 다양한 장치라는 부분을 분리된 계층으로 만들어서, 상위의 계층에서는 어떤 장치로 문자열이 표시가 되는지를 몰라야 할 수도 있다. 물론, 이것을 구현하는 방법은 다양하게 있지만, 여기서는 간단히 조건부 컴파일을 이용하도록 하겠다(사실 조건부 컴파일을 이용하는 방법을 권장하고 싶지는 않다.).

```

#include <stdio.h>

static void print_message(char *message) {
#define __CONSOLE__
    printf( "%s\n", message );
#endif
#define __NULL__
    /* Nothing to print */
#endif
#define __FILE_OUT__
    FILE *handle;
    if (( handle = open("output.res", "a" ) ) != NULL )
    {
        fprintf( handle, "%s\n", message );
    }
    fclose( handle );
#endif
}

int main(int argc, const char* argv[]) {
    print_message("Hello, World!!!\n");
    return 0;
}

```

위의 코드는 정의된(Defined) 값에 의해서 컴파일러가 어떤 장치를 출력으로 사용할 것인가에 따라, 문자열의 출력 결과가 어디로 갈지 정하도록 만든 것이다. 물론 위의 방법도 좋겠지만, 추상화라는 관점에서는 "print_message()"라는 이름과 함수의 내용이 아직 논리적인 비약이 있다. 즉, 추상화된 내용이 대표하는 것과 그 내용이 논리적으로 추상화 수준에서 차이가 조금 있다는 점이다. 따라서, 위의 경우를 좀 더 계층을 나누면 아래와 같이 된다.

```
#include <stdio.h>

static void print_message_to_console(char *message) {
    printf("%s\n", message);
}

static void print_message_to_null(char *message) {
    /* Nothing to do */
}

static void print_message_to_file(char *message) {
    FILE *handle;
    if ((handle = open("output.res", "a")) != NULL) {
        fprintf(handle, "%s\n", message);
    }
    fclose(handle);
}

static void print_message(char *message) {
#ifdef __CONSOLE__
    print_message_to_console( message );
#endif
#ifdef __NULL__
    print_message_to_null( message );
#endif
#ifdef __FILE_OUT__
    print_message_to_file( message );
#endif
}

int main(int argc, const char* argv[]) {
    print_message("Hello, World!!!\n");
    return 0;
}
```

이렇게 수정하고 나니, 추상화 수준에서는 계층적으로 함수의 이름과 실제 구현이 한 단계씩 더 구체화되는 방식으로 정의되고, 논리적인 비약도 함수의 이름과 구현 사이에서 사라졌다는 것을 볼 수 있다. 한 가지 문제는 계층화를 시키는 과정에서, 파라미터가 여러 함수를 거치면서 계속 전달되고 있다는 점과, 함수 호출의 깊이(Depth)가 깊어진다는 것이다. 얻게 되는 이점은 문제가 생기는 곳에 대해서 수정되어야 할 부분이 작은 부분에 집중 될 수 있다는 점과 기능을 확장할 때도 특정 부분에서만 변경이 발생한다는 것이다. 어떤 것을 더 중요하게 생각할지는 개발자의 몫이지만, 지금과 같은 간단한 코드가 아닌 경우에는 계층적인 접근이 유리하다는 것은 명확하다.

코드가 복잡해지면 고쳐야 할 부분이 어디에 있는지 확인하는 시간도 오래걸리며, 고치더라도 올바른 동작을 할 수 있는지 확인하는 과정도 필요하다. 가능한 수정해야 할 부분이 영향을 주는 부분을 최소화 시

키는 것이 해결 방법인 것이다. 간단한 코드에서 복잡한 코드로 진화하는 과정에서 위와 같은 추상화를 통한 계층적인 접근이, 구현의 의존성과 복잡성을 낮추는 훌륭한 도구임에는 확실하다.

[절차 지향과 객체 지향]

C언어는 절차지향 언어라고들 한다. 즉, 어떤 일을 하는 절차(Procedure)를 위주로 프로그램이 설계되고 구현된다는 것이다. 그렇다고, 이미 알고 있는 객체지향에 대한 개념을 포기할 필요는 없다(알고 있지 않다면 배워서 활용하면 된다). 객체지향의 원리는 상속이나 클래스에만 한정되는 것이 아니라, 사람이 생각하는 방식에 관련된 것이기 때문이다.

절차지향 언어는 일을 하는 절차를 중심으로 설계를 하게된다. 즉, 일을 하는 순서가 중요하다. 어떤 일 일 일어나고 다음에는 어떤 일을 해야 할지를 결정해야 한다. 예를 들어, 입력을 받아서 처리를 하고 출력을 하는 절차를 아래와 같이 프로그램 할 수 있을 것이다.

```
int main(int argc, const char *argv[]) {
    INPUT input;
    OUTPUT output;

    input = read_input();
    output = process_input(input);
    write_output(output);

    return 0;
}
```

즉, 순차적으로 일을 처리하는 절차를 가시적으로 알 수 있다. 객체지향 언어는 일을 하는 주체가 누구인가를 중심으로 생각한다. 따라서, 절차지향에서 중심으로 생각하는 "행위"보나는, 시스템을 구성하는 요소들이 무엇인지(역할)를 정의하고, 그들 간의 통신(책임)을 어떻게 할 것인가를 약속(Contract)하게 된다.

```
int main(int argc, const char *argv[]) {
    INPUT input;
    OUTPUT output;
    MESSAGE message;

    output.write(message.process(input.read()));

    return 0;
}
```

위의 코드에서 보듯이, “input”의 역할은 입력을 읽어내는 것이다. 처리를 담당하는 것은 “message”이며, 출력은 “output”이 담당한다. 즉, 각각이 명확한 자신만의 분리된 역할을 가지고 있으며, 역할에 따른 책임 범위도 명확하다는 것이다. 여기서 “input, message, output”을 객체라고 봤을 때, 각각이 협력을 통해서 문제를 해결하고 있다는 것을 알 수 있다.

얼핏 보면 차이가 나지 않는 것처럼 보이지만, 만약 요구사항이 “시스템에 입력을 읽고, 메시지를 생성해서 출력한다.”와 같이 주어진다면, 전자는 “입력”, “처리”, “출력”만 생각할 것이고, 후자는 “시스템”, “입력기”, “메시지”, “출력기”를 먼저 정의할 것이다. 즉, 같은 문제에 대해서도 시스템을 구성하기 위해서 어떤 일을 해야하는지 생각을 달리한다는 뜻이다.

어떤 언어가 좋고, 어떤 언어가 나쁘다는 것을 말하려는 것이 아니다. 사실 코딩의 "기본 원리"만 충실히 따르면, 어떤 언어를 사용하는지는 상관없이 충분히 좋은 프로그램을 만들 수 있다. 그 원리란 "의존성을 줄이고(Decoupling), 응집성(Cohesion)을 높인다"와 같이 단순하다. 하지만, 그 단순한 원리를 지키는 것은 어렵고 힘든 일이다. 어렵다는 이유는 코딩하는 전체 과정에서 계속 관점을 유지해야 하기 때문이다. 코딩에서는 원칙을 만들고 그것을 일관되게 유지하는 것이 정말 중요하다.

[C언어 표준에 대해서]

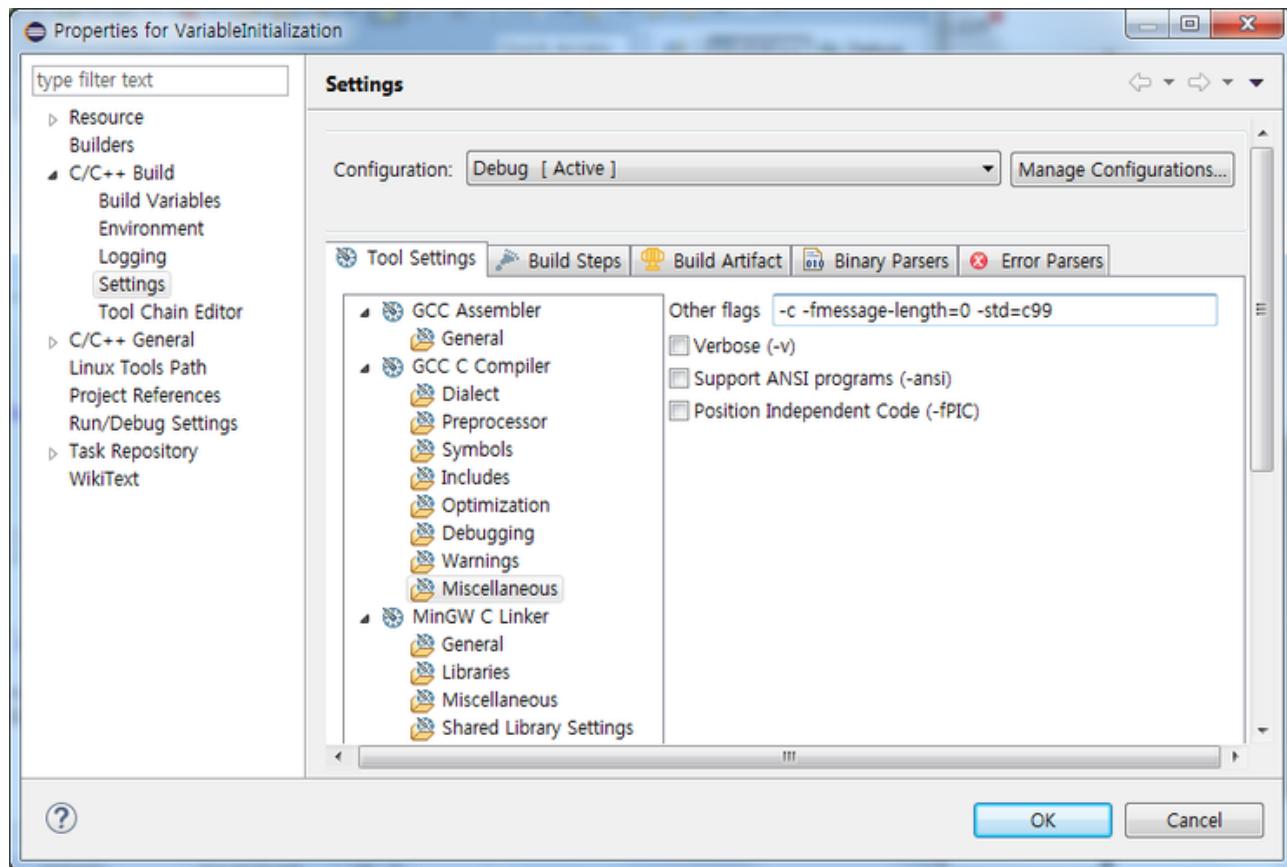
세상에는 다양한 C언어에 대한 컴파일러들이 존재하고, 소스코드의 호환성을 위해서는 표준이라는 것을 마련할 필요가 있었다. 컴파일러들이 호환이 되지 않는다면, 사용하려는 컴파일러별로 코드를 새로 작성해야 할 상황이 될 수도 있다. C언어도 표준이 존재한다. 그리고, C88, C99와 같이 제정된 년도를 사용해서 표준의 이름을 사용한다. 여기서는 간단히 자주 사용할 수 있는 한가지를 살펴보도록 하겠다.

변수는 사용되는 위치와 가깝게 정의될수록 이해하기 쉽다. 즉, 반복문 내에서의 변수 선언이나, 혹은 블록화된 코드에서 사용할 지역 변수를 정의하는 것이다. C99표준에서 새로 추가된 것으로, 이를 Eclipse에서 "mingw"를 이용해서 컴파일하기 위해서는 컴파일 옵션을 추가해 주어야 한다(최신 버전에는 달라질 수 있다).

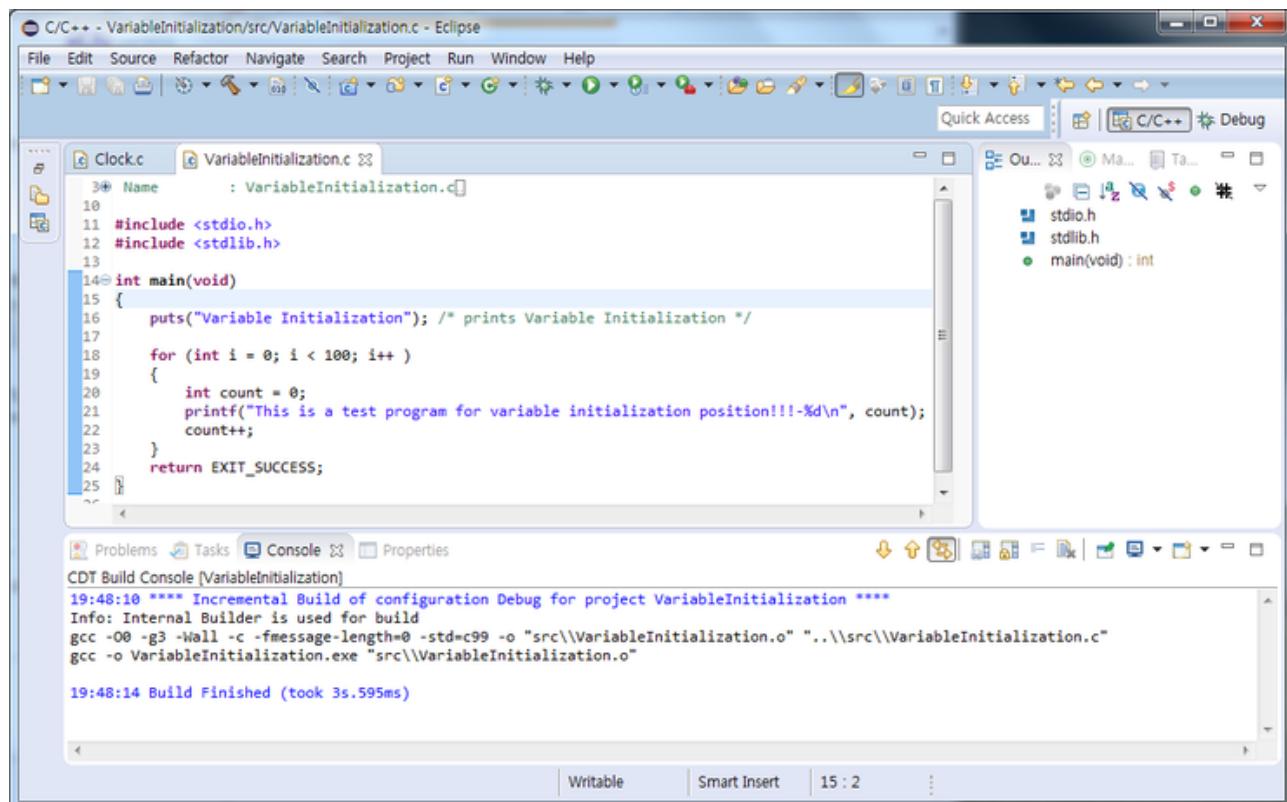
The screenshot shows the Eclipse C/C++ IDE interface. In the center, there is a code editor window displaying a file named 'VariableInitialization.c'. The code contains a main function that prints a message and a loop that prints a test program message. A build error is visible in the bottom right corner of the code editor. Below the code editor is a 'CDT Build Console' window showing the following output:

```
CDT Build Console [VariableInitialization]
..\src\VariableInitialization.c:18:2: note: use option -std=c99 or -std=gnu99 to compile your code
..\src\VariableInitialization.c:21:3: warning: too many arguments for format [-Wformat-extra-args]
    printf("This is a test program for variable initialization position!!!!\n", count);
    ^
01:47:15 Build Finished (took 216ms)
```

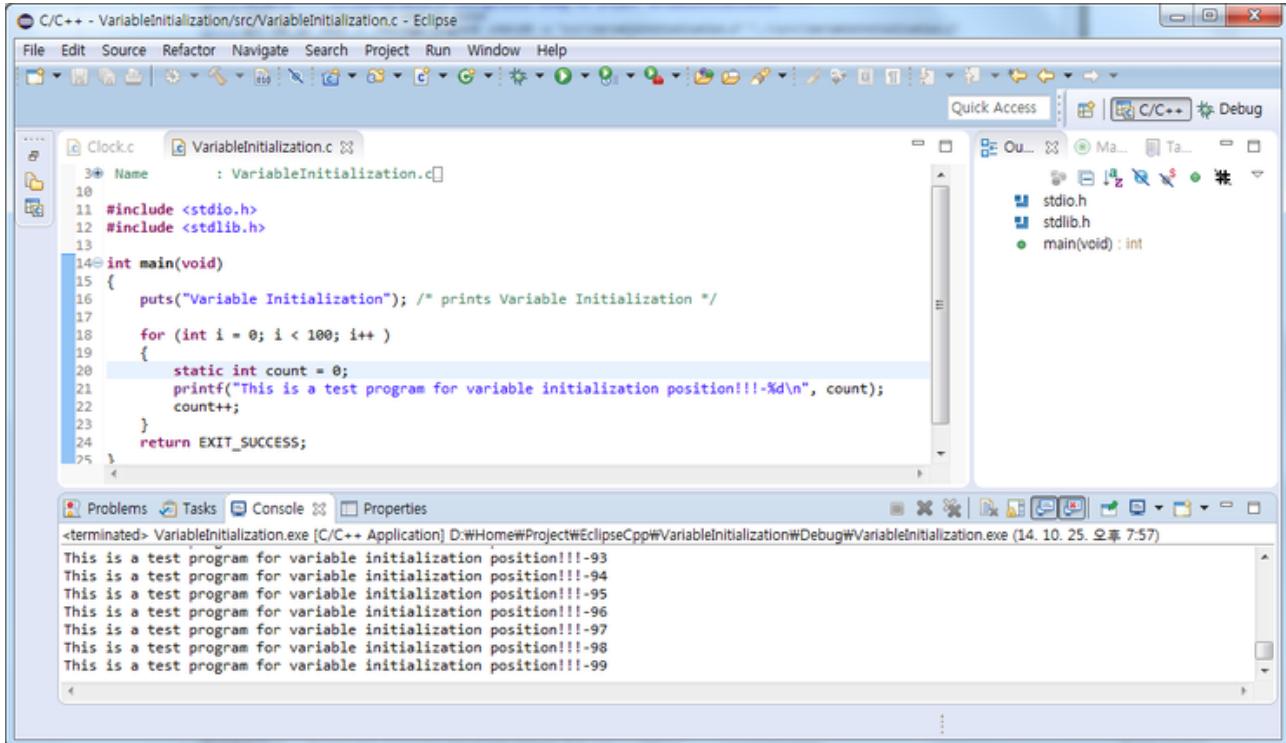
위의 그림은 C99 표준을 따르는 코드를 C99을 지원하는 옵션이 없이 컴파일한 결과다. 제대로 컴파일을 하기 위해서는 "-std=c99"나 "-std=gcc99"옵션을 컴파일러에 알려주어야 한다. 설정을 바꾸기 위해서는 Eclipse에서 "Project -> Properties -> Setting -> Tool Settings -> GCC C Compiler -> Miscellaneous -> Other flags"에 "-std=c99"을 추가해야 한다.



설정을 마치고 다시 컴파일을 하면 아래와 같이 보일 것이다. "C99" 표준으로 설정하기 전에 컴파일러가 오류(Error)를 표시하던 부분이 이제는 아무 문제없이 컴파일 될 것이다.



한가지 문제는 "for()"루프문 안에서 정의된 "count"변수는 매번 다시 정의된다는 것이다. 그래서, 실제로 실행하면 "count"값이 증가하지 않고, "0"으로 그대로 남는다. 이를 수정하기 위해서는 "static"을 "count"정의 앞에 추가해야 한다. 추가한 후에 다시 실행하면 아래와 같이 보일 것이다.



이제는 "C99" 표준을 지원하기 위한 Eclipse설정과 이에 따라 기존에는 사용할 수 없었던 문법을 사용할 수 있게 되었다. 변수는 사용되는 곳에 가까이 정의될 수록 더 이해하기 쉬운 코드를 만들 수 있으며, 이를 위해서 C99표준에 추가된 사항들을 알 수 있게 되었다.

물론, 다른 기능들도 있으니, 최신 C언어 표준을 읽어보기 바란다. 또한, 자신이 사용하는 컴파일러에서 지원하는 기능도 자세히 읽어봐야 할 것이다. 하지만, 일반적으로 컴파일러에 의존적인 코드 보다는 표준에 적합한 코드를 만드는 것이 더 좋은 코딩 방법이다. 작성된 코드는 어떤 컴파일 환경에서 사용될지 가정할 수 없으며, 환경에 의존적인 코드를 만들면 버그를 유발할 가능성도 높다(이것을 일일이 확인하기 어렵다면, 코딩 룰을 점검하는 도구를 사용하는 것도 한가지 방법이다.).

[C99표준의 printf() 함수 적용]

이전의 "printf()"함수의 제어 문자열에 대한 변경자(Modifier)에 컴파일 및 값의 범위 처리에 문제가 있었다. C99 표준에서는 "printf()"함수의 제어 문자열에 추가적인 변경자로 "l", "t", "z"등을 더 제공한다. "l"은 "unsigned long long int"에, "t"는 두 개의 포인터 간의 차를 구하고자 할 때, "z"는 "size_of()"의 복귀 값으로 사용하는 "size_t" 형을 위해서 존재한다.

```

#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE_OF_FIBONACCI 50

unsigned long long int storeFibonacciValue[MAX_SIZE_OF_FIBONACCI + 1] = { 0 };

unsigned long long int fibonacci(long nth) {
    if ((nth == 0) || (nth == 1)) {

```

```

        storeFibonacciValue[nth] = 1;
        return storeFibonacciValue[nth];
    }

    if ((storeFibonacciValue[nth - 1] != 0)
        && (storeFibonacciValue[nth - 2] != 0)) {
        storeFibonacciValue[nth] = storeFibonacciValue[nth - 1]
            + storeFibonacciValue[nth - 2];
    } else {
        storeFibonacciValue[nth] = fibonacci(nth - 1) + fibonacci(nth - 2);
    }
    return storeFibonacciValue[nth];
}

int main(void) {
    int i;
    long nth = 50;

    puts("This is Fibonacci Number Generation Project!!!\n");

    /* Run Algorithm */
    fibonacci(nth);

    /* Write Fibonacci numbers in order. */
    for (i = 0; i < MAX_SIZE_OF_FIBONACCI; i++) {
        printf(" %lu\n", storeFibonacciValue[i]);
    }
    puts("\n");

    return EXIT_SUCCESS;
}

```

위의 코드는 피보나치 수열을 재귀적인 호출과 동적인 프로그래밍을 동시에 이용해서 구하고 있다. 여기서 보고 싶은 것은 피보나치 값이 예전에 “printf()”함수로는 출력하기 어려울 때, C99표준에서 제공하는 포맷(Format) 문자열을 사용하는 방법이다.

이것도 앞에서 본 경우와 마찬가지로 오류(Error)메시지 없이 컴파일하기 위해서는 프로젝트 설정에서 컴파일 옵션에 “-std=c99”을 추가해 주어야 한다. 기본적으로는 C99표준으로 컴파일 하지 않기 때문이다.

[이름 없는(Anonymous) 구조체의 사용]

이름이 없는 구조체도 마찬가지로 C11 표준(2011년에 만들어진)에 추가된 것이다. 현재는 GCC에서 이미 지원하고 있지만, 컴파일러의 옵션을 “-std=c11”로 설정해 주어야 할 것이다. 이름 없는 구조체를 사용하는 이유는 구조체의 확장을 좀 더 쉽게하고, 이용도 쉽게 하기 위해서다. 아래의 예를 보도록 하자.

```

#include <stdio.h>
#include <stdlib.h>

struct named {
    unsigned int value;
    struct {

```

```

char *name;
unsigned age;
};

int main(void) {
    puts("Anonymous Structure Example 01");
    struct named myStruct = { 100, { "SH Kwon", 44 } };

    printf("The name : %s\n", myStruct.name);
    return EXIT_SUCCESS;
}

```

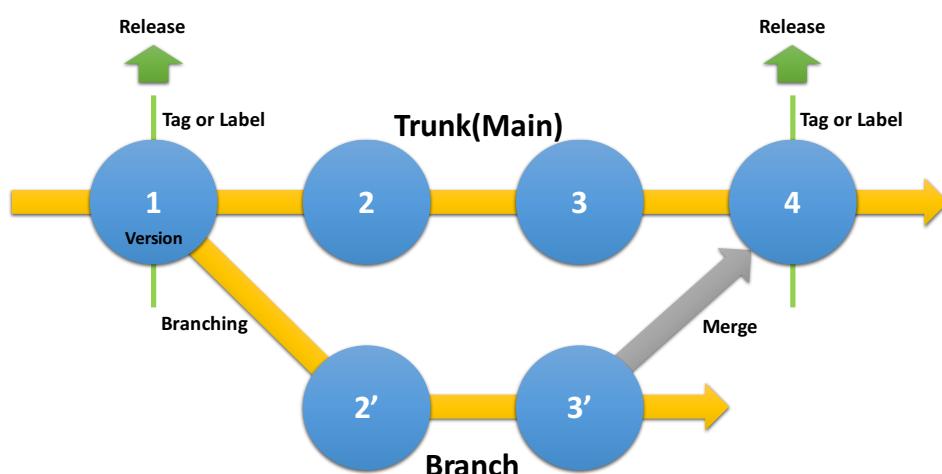
예제 코드는 이름이 있는 구조체 내에 이름이 없는 구조체를 정의하고 있는 것을 볼 수 있다. 즉, 내부에 선언된 구조체는 이름을 가지고 있지 않다. 초기화는 이름이 있는 구조체를 이용해서 일반적인 구조체를 초기화하는 방법을 사용하지만, 사용에서는 이름이 있는 구조체를 확장하는 것처럼, 내부의 구조체 필드를 직접적으로 사용할 수 있다("myStruct.name"처럼).

[버전 관리 시스템의 사용]

버전 관리 시스템은 개발자가 배포를 위해서 개발하는 소프트웨어의 변경을 기록하고 추적하는 시스템이다. 소프트웨어를 개발하는 회사는 모두 이런 시스템을 사용하고 있으며, 개발에서 반드시 사용해야 하는 필수적인 도구이다. 회사의 규모가 큰 곳에서는 상용 툴(Tool)을 많이 사용하고 있지만, 중소규모에서는 오픈 소스를 이용해서도 충분히 원하는 목적을 달성할 수 있다.

이런 버전 관리 시스템은 크게 두 가지가 있으며, 중앙 집중 방식과 분산 방식으로 나뉜다. 중앙 집중 방식은 모든 개발자가 하나의 저장소를 사용해서 개발하는 것으로 동일 장소에 있는 개발자들이 주로 많이 사용하는 방식이다. 분산 방식은 여러 곳에 분산된 개발자들이 사용할 수 있도록 만든 것으로, 저장소간의 자동화된 동기화를 제공한다.

분산 방식의 이점은 동시에 대규모 개발자들이 접속해서 개발을 진행할 수 있다는 점이며, 단점은 동기화하는데 걸리는 시간으로 인해 충돌이 발생할 가능성이 높다는 점이다. 즉, 한 쪽에서 작업한 코드가 원거리의 다른 저장소에서 작업한 코드간에 동기화가 맞지 않아서 발생하는 문제가 있을 수 있다. 중앙 집중 방식은 사용하기는 편하지만, 원거리에서 접속하는 개발자의 개발 속도가 느려지는 단점이 있을 수 있다. 따라서, 자신의 회사에서 어떤 식으로 개발할지 생각해보고, 작업의 특성에 맞게 도구를 선택하면 될 것이다.

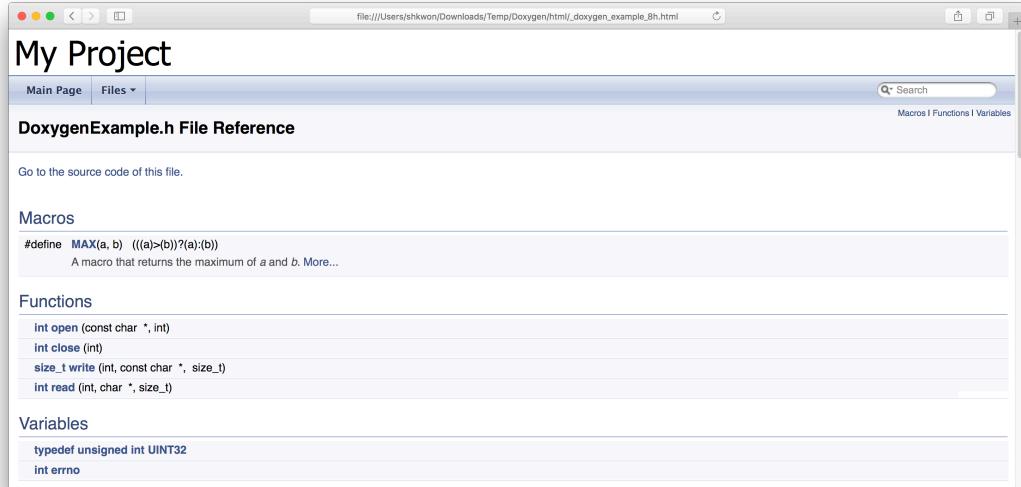


버전 관리 시스템은 변화를 데이터 베이스에 기록하는 방식으로 동작한다. 따라서, 저장소에서 가해졌던 모든 변화를 알고 있다. 변경이 발생했을 때, 이전과 달라 지는 부분이 어디며, 누가 무슨 이유(로그를 남길 경우)로 변경 했는지 보여 준다. 기존에 이런 시스템을 사용하지 않고 개발했을 때는 파일 서버를 설치해서 작업본을 날짜별로 복사해서 두는 방식을 사용했지만, 이것은 저장공간의 낭비와 더불어, 변경이 어디서 발생했고, 누가 변경 했는지를 알려주지 않았다(물론, 알아낼 수 있는 방법을 체계화 할 수도 있지만). 따라서, 이전 버전과 차이를 수동으로 다른 툴을 이용해서 검토할 수 밖에 없었다. 버전 관리 시스템이 주는 이런 혜택으로 인해서, 한 명 이상의 개발자가 관련된 과제에서는 반드시 사용하는 것이 좋다(개인이 혼자 개발해도 마찬가지다.).

저장소는 크게 3가지 종류로 나누어지며, 각각을 "Trunk, Branch, Tag(Label)" 등으로 나누어서 부른다. "Trunk"는 주요(Main) 버전으로 관리되는 코드를 말하며, "Branch"는 지선으로 파생된 버전을 말한다. "Tag(Label)"은 특정 시점에서의 "Snapshot(특정 시간에 만들어진 이미지)"을 말한다. 따라서, Trunk는 안정적인 버전을 유지하기 위해서 사용되며, Branch는 개발중인 새로운 기능들을 추가할 목적으로, Tag(Label)은 특정 버전을 배포할 목적으로 생성하게 된다. 추가적으로 개발자들 각각이 자신만의 개발 버전을 가지기 위해서 "Stream"을 만든 것도 가능하다. 이때는, 여러개의 Stream을 두고 병합(Merge)하는 절차가 필요하다(Perforce와 같은 툴은 이것을 지원한다).

[“Doxygen”을 이용한 코드의 문서화]

문서화는 언제나 프로그래머에게 부담이다. 코드를 만드는 일은 좋아하지만 문서를 쓰는 것은 싫어한다. 가능하면 다른 사람이 만들어주는 문서를 그냥 사용하거나, 혹은 조금만 고쳐서 사용하고 싶은 것이 문서화에 대한 일반적인 태도다. 하지만, 문서는 코드를 설명하기 위해서 필요하며, 특히 협업을 하는 경우에는 어떻게 코드가 만들어져 있으면, 어떻게 함수들을 사용할 수 있는지 알려주어야 할 필요가 있다. 이 때 사용할 수 있는 방법이 "Doxygen"과 같은 자동화된 문서화 도구를 이용하는 방법이다.



먼저, 문서화의 대상이 되는 것에는 어떤 것들이 있을까? 즉, 문서를 읽는 사람의 입장에서 어떤 것들을 알고 싶어할까? 가장 먼저 필요한 것은 역시 제공되는 함수에 대한 사용법이다. 그리고, 가장 좋은 문서화 방법은 사실 함수를 사용하는 많은 테스트 케이스(Test Case)를 제공하는 것이다. 하지만, 그렇다고 완전하지는 않다. 이때 주석(Comment)를 활용해서 코드를 문서화 시키는 방법을 사용하게 된다. Doxygen은 문서를 읽어서 주석 부분을 해석해서 필요한 문서를 생성해 낸다. 따라서, Doxygen에서는 주석을 어떻게 사용하는지 이해할 필요가 있다.

함수를 문서화 하기 위해서는 "함수의 이름과 사용되는 인자들에 대한 설명, 돌려주는 값에 어떤 것"들이 있는지를 알아야 한다. 추가적으로 함수가 인자에 사용하는 자료구조에 대한 것들도 알고 있으면 좋겠지

만, 너무 상세한 정보까지 일일이 다 기술하는 것은 함수를 사용하는 측면에서는 별로 달가울 것이 없다. 즉, 너무 많은 정보는 오히려 사용하는 측에서는 부담이 된다는 것이다. 따라서, 가능한 정보를 담을 수 있는 자료구조를 정의하고, 그것에 대한 직접적인 조작은 막는 것이 좋다.

Doxxygen을 이용할 때의 주석은 여러 형태가 있지만, 주로 C 스타일에서는 "/* ... */"나 /*! .. */와 같은 것을 사용한다. 아래의 예를 잠시 보도록 하자(이것은 Doxygen의 제공되는 문서를 인용했다.).

```
/*! \file structcmd.h
\brief A Documented file.
```

Details.

*/

```
/*! \def MAX(a,b)
\brief A macro that returns the maximum of \a a and \a b.
```

Details.

*/

```
/*! \var typedef unsigned int UINT32
\brief A type definition for a .
```

Details.

*/

```
/*! \var int errno
\brief Contains the last error code.
```

\warning Not thread safe!

*/

```
/*! \fn int open(const char *pathname,int flags)
\brief Opens a file descriptor.
```

\param pathname The name of the descriptor.

\param flags Opening flags.

*/

```
/*! \fn int close(int fd)
\brief Closes the file descriptor \a fd.
\param fd The descriptor to close.
*/
```

```
/*! \fn size_t write(int fd,const char *buf, size_t count)
\brief Writes \a count bytes from \a buf to the filedescriptor \a fd.
\param fd The descriptor to write to.
\param buf The data buffer to write.
\param count The number of bytes to write.
*/
```

```
/*! \fn int read(int fd,char *buf,size_t count)
```

```
\brief Read bytes from a file descriptor.
\param fd The descriptor to read from.
\param buf The buffer to read into.
\param count The number of bytes to read.
*/
```

```
#define MAX(a,b) (((a)>(b))?(a):(b))
typedef unsigned int UINT32;
int errno;
int open(const char *,int);
int close(int);
size_t write(int,const char *, size_t);
int read(int,char *,size_t);
```

여기서는 자세한 Doxygen 주석 작성 방법에 대해서는 이야기 하지 않겠다. 필요하다면 Doxygen을 직접 설치해서 관련된 문서를 읽고 실행해 보기 바란다. 간단히 자주 사용하는 것만 이야기하도록 하겠다.

Doxygen 주석을 적용하는 대표적인 파일은 주로 타입이나 함수를 정의하는 헤더(Header) 파일이다. 헤더 파일은 다른 코드를 구현할 때 C에서 포함(Include)되어야 하기에, 이곳에 적어두면 문서화된 자료를 제공할 때 유용하다. 즉, 코드를 라이브러리 형태로 배포하더라도 헤더에 대해서는 따로 배포해야하기 때문이다. 그리고, 외부에서 필요한 자료구조나 함수들을 정의하고 있기에, 내부 구조를 보여주지 않아도 된다.

```
/*! \fn int read(int fd,char *buf,size_t count)
\brief Read bytes from a file descriptor.
\param fd The descriptor to read from.
\param buf The buffer to read into.
\param count The number of bytes to read.
*/
```

함수에 대한 주석을 위해서 "\fn"을 사용하고 있으며, 함수의 정의를 그대로 가져다 두었다("int read(...)). "\brief"는 함수가 어떤 일을 하는지 간단히 설명할 때 사용한다. "\param"은 함수의 파라미터 각각에 대해서 하나씩 한 줄을 사용했으며, 각각의 파라미터가 어떤 역할을 하는지 알려주고 있다. 추가적으로 "\param" 다음에 "[in], [out], [in,out]"을 사용하면, 각각의 파라미터가 입력, 출력, 입/출력으로 사용된다는 것을 표현할 수 있다.

이곳에서는 Doxygen의 주석 문법을 다 훑어보지는 않았다. 더 필요한 정보가 있으면 직접 Doxygen을 설치해서 확인하기 바란다. 아래는 Doxygen을 다운로드 받을 수 있는 곳이다.

<http://www.stack.nl/~dimitri/doxygen/>

추가적으로 호출 그래프(Call Graph)등을 보기 위해서는 "Graphviz"와 같은 프로그램이 필요할 수 있다. 이런 부분들도 Doxygen 문서를 참고하기 바란다. 참고로 "Doxygen Wizard"는 명령 라인(Command Line)에서 사용하는 불편을 개선하기 위한 사용자 인터페이스를 제공하는 프로그램도 있다.

Doxygen은 단순히 문서화만 시키는 도구가 아니며, 사용하기에 따라 소스 코드를 분석하는 목적으로도 사용할 수 있다. 즉, 전체 구조를 파악하는데도 유용하며, 웹 브라우저로 HTML을 이용해서 코드를 추적하고 이해하는데도 활용 할 수 있다.

[UML을 이용한 코드 실행의 분석]

UML(Unified Modeling Language)는 소프트웨어 설계(모델링)를 위해서 사용하는 언어다. 다양한 다이어그램(Diagram)으로 프로그램이 어떤 구조를 가지고 있으며, 어떻게 실행되고, 어떤 상태를 가지고 있는지 표현할 수 있다. “PlantUML”과 같은 툴은 텍스트 입력을 받아서 UML 다이어그램을 생성할 수 있도록 만들어주며, UML 다이어그램을 그리기 위한 자신만의 고유한 언어를 정의하고 있다. “PlantUML”을 사용해서 그림을 그리려면, “Graphviz”와 같은 툴도 함께 필요하다. 각각의 툴은 아래와 같은 곳에서 다운로드 받을 수 있다.

- PlantUML의 다운로드 주소 -> <http://plantuml.com/download>
- Graphviz의 다운로드 주소 -> <http://www.graphviz.org/Download..php>

Eclipse는 “PlantUML” 플러그인이 있으며 설치하기 위해서는 아래의 사이트를 “Install New Softwares”에서 추가해 주어야 한다.

- PlantUML Eclipse 플러그인 설치 주소 -> <http://files.idi.ntnu.no/publish/plantuml/repository/>

Eclipse 플러그인 형태로 사용하기도 하지만, 이곳에서 보여주고 싶은 것은 코드를 실행해서 UML의 “시퀀스(Sequence)” 다이어그램을 생성하는 것이다. 다음의 예제 코드를 보도록 하자.

```
#include <stdio.h>
#include <stdlib.h>

#define UML_START      printf("@startuml\n") /* UML 다이어그램을 위한 언어 시작 */
#define UML_END        printf("@enduml\n")   /* UML 다이어그램을 위한 언어 끝 */
#define UML_CALLER(x)  printf("%s -> ", (x)) /* 호출한 모듈의 이름을 쓴다 */
#define UML_CALLEE(x)  printf("%s : %s()\n", (x), __FUNCTION__ ) /* 호출된 모듈의 이름
과 함수를 쓴다. */

void function_C(void) {
    UML_CALLEE("Module_C");
    return;
}

#define MAX_CALL_COUNT 10

void function_B(void) {
    UML_CALLEE("Module_B");
    UML_CALLER("Module_B");
    function_C();
}

void function_A(void) {
    UML_CALLEE("Module_A");
    UML_CALLER("Module_A");
    function_B();
}

int main(void) {
    puts("PlantUML Example 01"); /* prints PlantUML Example 01 */
}
```

```

UML_START;
UML_CALLER("Main");
function_A();
UML_END;
return EXIT_SUCCESS;
}

```

이 코드는 단순히 함수의 호출 시에 어떤 함수를 호출하고 있으며, 누가 호출 했는지를 알기 위한 “PlantUML” 언어를 출력하는 것이다. 실행하고나면 아래와 같은 결과를 출력으로 화면에 표시하게 된다. 표시되는 내용이 PlantUML에서 사용하는 다이어그램을 위한 언어다.

[결과]

```

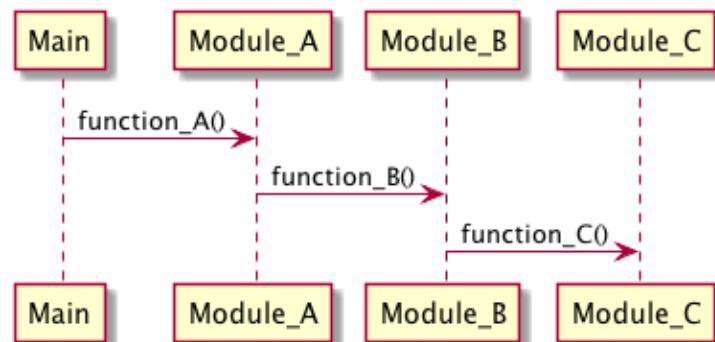
@startuml
Main -> Module_A : function_A()
Module_A -> Module_B : function_B()
Module_B -> Module_C : function_C()
@enduml

```

이 결과를 파일로 저장 했다면, 다운로드 받은 “plantuml.jar”을 이용해서 다이어그램으로 변환할 수 있다. 커맨드 라인에서 사용 방법은 아래와 같다.

```
>java -jar plantuml.jar input.txt
```

“input.txt”는 변환할 UML 다이어그램의 텍스트를 가지고 있는 파일이다. 변환한 결과는 그림 파일의 형태로 저장된다(“input.png”가 될 것이다.).



생성된 다이어그램에서 보듯이, 코드의 실행 경로가 어떻게 되는지 실제 실행을 통해서 자동으로 생성해 낸다. 즉, 생성된 다이어그램을 통해서 코드의 동작이 설계된데로 동작 하는지 확인할 수 있다. 혹은, 이미 존재하는 코드에서 이해를 돋기 위한 그림을 뽑아낼 목적으로도 사용할 수 있다(코드에서 설계를 뽑아낼 경우).

[좋은 코드의 특성]

기초를 배우는 사람들은 어떤 코드가 좋은 코드인지 반드시 알아야 한다. 기초가 튼튼해야 나중에 무너지는 경우가 생기지 않는다. 소프트웨어 개발에서 말하는 “무너짐”은 “초과되는 비용”과 “개발 일정의 지연”, “잦은 품질 문제”이다. 그리고, 결과적으로는 그런 소프트웨어는 시장에서 살아남지 못한다. 좋은 코드는 그렇지 못한 코드보다 오래 살아남는다. 그래서, 생각보다 유지보수(Maintenance) 비용도 더 들기도 한다. 하지만, 그런 비용을 다 합한다 해도 좋은 코드가 벌어들이는 이익 보다는 많지 않을 것이다. 좋지 못한 코드는 벌어들이는 이익보다 비용이 지속적으로 더 빨리 불어난다. 따라서, 회사의 경쟁력은 시간이 흘러감에 따라 점차 낮아지는 것이 일반적이다.

코드 문제로 인해서 발생하는 비용은 생각보다 훨씬 많다. 수정(기능 추가 혹은 버그 해결)에 필요한 소프트웨어 개발자와 테스터의 인건비를 합쳐야 하며, 신규 과제의 지연으로 인해 발생하는 기회 손실 비용 까지도 포함되기 때문이다. 따라서, “좋은 코드 작성”의 실패로 인한 비용은 눈으로만 파악할 수 있는 것이 아니다. 다음은 좋은 코드의 특성들을 나열한 것이다.

01. 읽기 쉽다.

; 가장 중요한 특징으로 다른 사람이 읽기 쉽게 작성된 코드가 좋은 코드이다. C언어를 위해서 필요한 것이 아니라, 모든 코드는 반드시 읽기 쉬워야 한다. 읽기 어려운 코드를 짜는 것은 버그 수정이나 신기능 추가와 같은 유지보수에 치명적인 영향을 주게되기 때문이다. 나 보다는 남을 위해서 일하는 것이 결국 자신을 위한 것이다. 함수의 이름이나 변수의 이름(함수내의 지역 변수의 이름도 포함하고, 반복문에 사용되는 임시 변수 들도 이름을 가지는게 좋다. 예외적으로 i, j, k등과 같은 배열을 접근하기 위한 임시변수는 이미 관례적(Conventional)으로 많이 사용 되기에 특별한 이름을 가질 필요가 없다.)

02. 개념적으로 분리되어 있다.

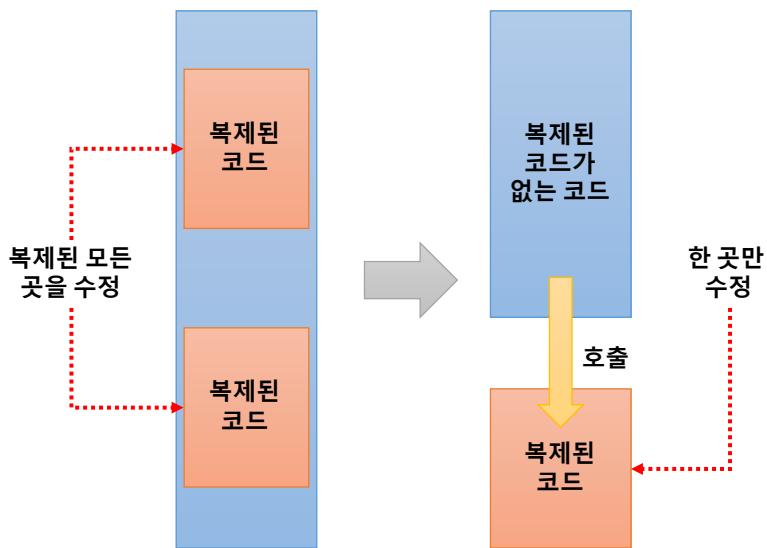
; 모든 코드 조각(Unit)들이 하나의 개념을 구현한다. 여러 가지 일을 한번에 하지 않으며, 추상적으로 한 가지 개념만 구현한다. “개념적으로 한 가지 일만 한다”는 의미는 함수가 하는 일을 하나로 정의할 수 있다는 말이 되며, 그 정의를 이용해서 함수의 이름을 정한다. 특히, C언어에서는 함수를 가장 작은 독립적으로 실해할 수 있는 단위로 보기에 이름을 정하는 것은 중요한 일이다. 함수의 계층을 따라 내려갈 경우, 상위 함수가 가지는 개념의 한 수준 아래의 이름을 하위 함수가 가져야 한다.

03. 변경에 대한 대비가 되어 있다.

; 변경할 수 있는 부분들을 미리 설계에 반영해서 구현되어 있다. 변경이 생기더라도 지역적(Local)인 영향만 준다. 변경에 대한 대비는 주로 인터페이스의 정의로부터 시작한다. 좋은 함수를 만드는 것은 항상 이런 변경에 대비한 한가지 방법이다. 설계의 마지막도 그러한 인터페이스에 대한 정의다. 인터페이스의 정의가 완료되고, 이를 이용해서 구현할 내용이 명확해지면 코딩할 수 있는 준비가 완료된 것이다. 하지만, 처음부터 어디를 변경 포인트(Point)로 정할 것 인가를 정하는 것은 쉽지 않다. 따라서, 새로운 기능의 추가나 구조의 변경이 있을 때는 변경 부분에 대한 고려도 다시 한번 해야 할 것이다.

04. 주석이 적다.

; 대부분의 개발자가 주석을 많이 사용하는 이유는 코드를 설명하기 위한 것이지만, 코드 자체가 쉽게 이해 된다면 주석을 작성할 필요가 없다. 주석이 필요한 경우는 선택에 대한 이유(Why?)를 알려줄 경우다. 실무자들은 코딩 할 때 한 라인당 하나의 주석을 달라고 할 정도로 많이 주석을 넣지만, 코드가 쉽게 이해가 되는데도 그 코드를 다시 설명하는 주석을 많이 사용한다. 이것도 일종의 반복(Repeat)이며 낭비다, 일반적으로 소프트웨어 개발에서는 반복은 없는게 좋다. 또한, 주석은 작성은 쉽게 할 수 있을지 모르지만, 유지하는 것은 코드를 관리하는 것 만큼의 노력이 필요하다. 따라서, 주석으로 이해하기 어려운 코드를 설명하기 보다, 주석이 없이도 이해가 쉬운 코드를 만드는 것이 더 효과적인 방법이다.



05. 중복이 적다.

; 중복된 코드들이 많은 경우는 정리되지 않고 급하게 개발된 코드인 경우가 많다. 중복은 최소화 되어야 하며, 가능하다면 없애야 한다. 코드의 중복도 있지만, 논리의 중복도 확인해야 한다. 특히, "switch()"문과 같은 경우, 각각의 경우에 대해서 반복적으로 코드가 있는 경우가 많이 생기며, 또한 여러 곳에서 비슷한 방식으로 사용할 코드를 복사해서 붙여넣는(Copy & Paste) 경우도 빈번하게 발생한다. 이런 것들은 코드가 변경될 경우 일일이 찾아가며 수정해야 하는 어려움이 있다. 반복되는 코드는 함수로 만들어서 모아 두어야 하며, 고쳐야 할 경우 한 곳에서만 수정할 수 있도록 만들어야 한다.

06. 마치 한 사람이 개발한 것처럼 보인다.

; 여러 사람이 코딩에 참여 했지만, 개발 결과물에 대해 관리를 잘 했다는 의미다. 즉, 규칙(Rule)(혹은 규정)에 기반해서 코딩이 진행 되었으며, 코드들이 전부 리뷰(Review)가 되었을 가능성이 높다. 이러한 규칙을 코딩 룰(Coding Rule, Coding Convention, Coding Standard)이라고 부르며, 보통의 경우 회사 차원보다는 팀 차원이나 과제 차원에서 관리된다. 과제에 참여하는 모든 개발자들은 그 내용을 잘 숙지하고 있어야 하며, 일관되게 지속적으로 사용하는 것이 중요하다. 코드를 꾸미는 것이 무의미 하다고 이야기 하는 사람들도 있지만, 이해를 돋기 위해서 코드를 꾸며주어야 할 필요도 있다. 물론, 그렇다고 지나치게 화려하게 만들 필요도 없다.

07. 자동화된 테스트가 존재한다.

; 자동화된 테스트가 없다면 변경이 발생할 경우 매번 오랜 시간을 테스트로 낭비하게 된다. 자동화된 테스트는 개발자가 만들어야 할 필수 산출물이다. 그리고, 그 결과로 만들어지는 테스트 범위(Coverage)는 얼마나 코드가 테스트 되었는지를 알려주는 척도로 사용할 수 있다.(100% Coverage라고 해도 버그는 존재한다.) 하지만, 실제 개발에서는 자동화 된 테스트는 제한적으로만 사용될 뿐이다. 단위 테스트와 같은 것은 거의 하지 못한다. 자동화된 테스트가 있다면, 개발자는 자신의 코드에 대해서 확신을 가질 수 있지만, 그렇지 않다면 언제나 불안한 상태에서 변화를 거부하게 된다("동작하는 코드는 바꾸지 않는다"는 원칙이 있다.).

08. 코드속에 모듈간의 상하관계가 명확하다.

; 상식적으로 복잡한 문제는 내부에 계층적인 구조를 가질 가능성이 높다. 소프트웨어는 복잡한 문제를 어떻게 다루느냐에 따라 품질이 달라질 수 있으며, 일반적으로 계층적(Layering)으로 문제를 해결해 간다. 이때, 상위 계층은 바로 아래의 하위 계층에 의존적이며, 하위 계층은 상위 계층에 의존적이지 않다(즉, 호출이 없다. 하지만, 상위 계층에서 하위 계층을 호출할 때 필요한 인터페이스는 하위 계층에서 제공해주어야 할 의무가 있다.).

09. 함수의 인자 개수가 적다.

; 한 덩어리로 개발되지 않은 코드들은 당연히 다른 함수(혹은, 메쏘드:Method)에 대한 호출을 가진다. 그런 함수들의 정의가 한 가지 역할에 충실히 되어 있다면, 인자들의 갯수는 적을 가능성이 높다. 많다고 하더라도 관련된 데이터들을 사용자 정의 자료형으로 만들어서 함수와 같이 관리할 수 있을 것이다. 함수의 인자 개수가 많다는 것은 함수가 여러가지 일을 하고 있을 가능성이 높다. 즉, 다루고 있는 데이터들이 서로 상관 관계가 멀다는 의미와 같다. 이런 경우에는 각각의 데이터에 대해 분리된 함수를 제공할 수도 있다.

10. 모든 함수의 호출 뒤에는 반드시 오류를 검증하는 코드가 있다.

; 함수들은 실패할 가능성이 항상 존재한다. 따라서, 반드시 호출 결과를 검증하는 코드를 동반한다. 완전한 코드가 없듯이, 자신이 만든 함수를 남들이 사용하더라도 최소한 사용법과 복귀 값에 대한 설명은 필요하다. 복귀값이 필요없는 함수를 만들수도 있다. 하지만, 디버깅 하는 과정에 있다면, 함수의 호출 결과를 외부에서 검사할 수 있도록 만들어주는 것이 도움이 된다. 항상 남의 코드를 전적으로 믿지말고 방어적인 자세로 코딩하는 것이 좋을 것이다.

11. 매직 넘버(Magic Number)가 없다.

; 아마도 코드를 읽을 때 가장 어려운 부분이 매직 넘버일 것이다. 왜 그 수가 나왔는지, 그리고 어떤 의미인지에 대해서 전혀 아무런 답이 없는 코드들이 많다. 코드에서 그런 수를 만날 때면 대부분 그냥 그려려니하고 넘어가게 되지만, 결국에는 다시 그 부분으로 가서 해석하는데 많은 시간을 보내고 만다. 매직 넘버를 줄이는 방법은 "enum"과 같은 것을 사용하거나, 상수 변수(const variable)를 정의해서 사용하도록 한다. 매크로를 이용할수도 있지만, 디버거에서 사용할 때는 변수 이름을 주는 것이 도움이 된다.

```
#define DEBUG
void function(unsigned int param) {
#if defined(DEBUG)
    if ((param <= MIN) || (param >= MAX)) {
        printf("Error!!!\n");
        exit(-1);
    }
#endif
    /* Do Something Here!!! */
    return;
}
```

12. 함수의 내부에는 넘겨받은 인수에 대한 적합한 범위를 검증한다.

; 방어적인 코딩을 할 경우, 함수가 넘겨받는 인수가 적합한 범위를 가지는지 검사해야 한다. 누가 어떻게 사용할 지 모르는 코드이기 때문이다. 특히, 배포와 디버깅을 할 경우를 대비해서, 이런 코드들을 따로 끌어서 조건부 컴파일을 시킬 수도 있다(즉, 검사가 완료되어 더 이상 필요가 없을 경우에는 배포하는 이미지에서는 제거한다). 혹은, 그냥 코드에 남겨서 잘못된 호출이 발생한 원인을 남기고 프로그램을 종료 시킬 수도 있다.

이곳에서 나열한 것 이외에도 다양한 것들이 있을 수 있지만 일단은 이 정도로 만족하도록 하자. 중요한 것은 어떤 규칙을 가지고 지속적으로 코드를 개선하는 것이다. 실무에서 만나는 코드들은 위와 같지 않으며 이해하기도 쉽지 않다. 특정 도메인(Domain)에 대한 지식을 요구하는 경우도 많으며, 그런 것들이 누락된 상태에서 코드를 읽어서 정보를 얻는 것은 다양한 지식과 경험을 요구한다. 따라서, 우리가 해야 할 일은 그런 시간을 단축시켜줄 수 있는 방법을 고안하고 코딩에 적용하는 것이다.

[관리하기 쉬운 코드를 만드는 원리]

프로그래밍의 기본적인 원칙은 "의존성 제거(Decoupling)와 높은 응집성(Cohesion)"으로 요약된다. 의존성은 어쩔 수 없이 발생할 수 밖에 없지만 가능하다면 줄여야 한다. 응집성은 코드의 조각들(모듈 단

위, 파일 단위, 함수나 클래스 단위에서도)이 주어진 역할만 충실히 수행하도록 뭉쳐져야(함께 관리되어야) 한다. 의존성이 줄어든 코드는 의존하고 있는 것들이 변경 되더라도 최소한의 영향만 받게된다. 따라서, 의존성을 줄이면 확장이나 변경이 쉬워진다. 응집성이 높은 코드는 외부에 대한 의존성이 줄어들게 되며, 잘 정의된 역할(Role)만을 수행하기에 변경에 대한 영향을 외부로 전달하지 않고 내부에서 해결한다. 이 두 가지가 코딩을 지배하는 기본 원리다.

객체지향 언어를 사용하는 사람은 한번쯤은 "SOLID"라는 원칙(Principle)을 들어봤을 것이다. 즉, 객체지향 코딩에서는 SOLID 원칙에 충실히 코딩을 해야 앞에서 이야기한 의존성이 낮고 응집성이 높은 코드를 만들어낼 수 있다. SOLID원칙은 "SRP(Single Responsibility Principle), OCP(Open Close Principle), LSP(Liskov Substitution Principle), ISP(Interface Segregation Principle), DIP(Dependency Inversion Principle)"을 의미한다. 이런 원칙들이 객체지향 언어에서만 유용한 것은 아니며, C언어와 같은 절차지향 언어에서도 의존성을 줄이고 응집성이 높은 코드를 만드는데 도움을 줄 수 있다.

- **SRP(Single Responsibility Principle)**은 한 가지 책임(역할)만을 가지도록 코딩하라는 말이다. 여기서 말하는 한가지 책임이란 추상적인 수준에서 한 가지 개념을 구현하는 코드를 말한다. 책임이란 결국 역할에 대해서 주어지는 활동의 범위를 말하며, 그 범위를 표현할 수 있는 하나의 추상적인 개념을 찾을 수 있으면 된다. 예를 들어, 운영체제를 생각해 보면 큰 개념에서는 하드웨어 자원을 관리하는 소프트웨어라는 것으로 표현되며, 다시 그 개념을 나누면 CPU 스케줄링, 메모리 관리, 파일 시스템, 네트워크로 나누어 진다. 각각이 하나의 추상적인 개념을 구현하고 있으며, 다시 하위 개념으로 쪼갤 수 있다. 이때 각각의 개념은 자신이 해야할 활동의 범위가 있다는 것을 알 수 있다. 즉, 그 범위의 역할만 충실히 코드를 만들면 된다.

C언어에서 가장 작은 독립된 실행 단위(Unit)가 함수라고 할 때, 하나의 함수는 하나의 개념을 구현해야 한다. 함수들이 모인 파일은 한 가지 개념을 구현하기 위한 함수들로 구성되어야 하며, 파일들을 묶은 폴더(디렉토리, 모듈)들은 한 가지 개념을 구현하기 위해서 필요한 파일들로 묶여야 한다. 다시 이런 폴더들이 모여서 하나의 서브시스템을 구현하고, 서브시스템들이 모여서 시스템을 구현하게 된다. 묶이는 과정에서 개념적으로 생략(혹은, 지나친 비약)이 발생한다면, 더 세분화할 것이 남았다는 이야기다. 따라서, 이때는 좀 더 추가적으로 정의할 개념들이 있는지 분석해야 한다.

- **OCP(Open Close Principle)**은 확장에 대해서 열려(Open)있고, 변경에 대해서 닫혀(Close)있어야 한다는 원칙이다. 이 원칙은 지속적인 변경이 발생하는 소프트웨어의 특성에 대해서 어떻게 대응할 수 있을 것인가에 대한 해결책이다. 즉, 새로운 기능을 구현하기는 쉬우면서도, 변경의 영향이 부수효과(Side Effect)가 없도록 해야한다는 말이다. 이러한 코드를 만들기 위한 핵심은 추상화(Abstraction)에 있다. C++언어와 같은 객체지향 언어에서는 ADT(Abstract Data Type)을 제공하며, 확장을 위해서 상속과 같은 기법을 사용할 수 있다. 또한, 상속된 클래스도 부모 클래스가 제공하는 인터페이스를 제공하기에, 부모 클래스에 접근하듯이 상속된 자식 클래스를 접근할 수 있도록 하고 있다.

C언어에서도 추상화를 통해서 이와 유사한 기능을 제공할 수 있다. 즉, 자료구조에 대한 상세한 내용은 가지고, 단순히 자료구조의 형(Type)에 대한 정보만을 제공한 후, 이 자료구조를 접근하는 함수들을 제공하는 방법이다. 호출하는 코드에서는 상세한 자료구조에 대한 정보를 알 수 없기에 반드시 함수를 통해서 접근해야 한다는 단점이 있지만, 내부적인 자료구조의 변경이나 함수 코드의 수정이 호출하는 코드에 대한 영향을 최소화 시킬 수 있는 방법이다. 호출하는 코드와 호출을 받는 코드는 서로 모르면 모를 수록 더 좋은 코드가 만들어질 가능성이 높다. 너무 많은 정보를 공개하는 것은 코드들의 의존성을 높이기 때문이다.

- **LSP(Liskov Substitution Principle)**은 대체(Replacement)에 대한 것이다. 즉, 프로그램에서 필요한 객체(Object)를 하위의 타입으로 프로그램의 정확성을 해치지 않고 교체가 가능해야 한다는

원칙이다. 객체지향 언어의 경우 클래스를 정의하고, 이를 상속한 클래스들이 부모의 역할을 대신 처리할 수 있다는 것으로, 객체를 접근하는 쪽에서는 접근되는 객체의 타입만을 알고 있을 뿐이다. 실제 어떤 객체를 접근하는지 모르기에, 상속받은 객체들은 어떤 것이라도 부모의 역할을 대신할 수 있다. 이것은 실행 중에 프로그램의 동작을 변경할 수 있는 방법을 제공하기에, 호출하는 코드의 변경없이 새로운 환경에 동적으로 적용할 수 있도록 만들어준다.

C언어에서는 이와 비슷한 예로 다양한 파일 시스템을 하나의 함수로 접근하는 경우를 들 수 있다. 즉, 하위에서 실제로 구현하는 방법은 다르지만, 상위의 코드는 일관된 인터페이스를 사용해서 각각의 전혀 다른 파일 시스템을 사용할 수 있다. 변경은 하위 계층의 코드에서만 일어나게 되며, 상위 계층 코드의 변경없이 새로운 파일 시스템을 지원할 수 있다. 이를 위해서는 C의 "struct"와 "*pointer"를 이용해서, 동적으로 변경 가능한 코드를 구현하는 방법이 있다. 포인터는 주소를 가르키기에 변수의 주소나 함수의 주소에 상관없이 사용할 수 있으며, 이를 잘 이용하면 동일한 인터페이스를 가지지만 새로운 기능을 추가하는데 유용하게 활용할 수 있다.

- **ISP(Interface Segregation Principle)**은 필요한 인터페이스만 관심을 두어야 한다는 것이다. 즉, 호출하는 측에서 필요로 하는 인터페이스들을 나누어서 관리하라는 말이다. 상호 관련성이 없는 인터페이스들을 묶어서 하나로 만들어 두면, 인터페이스를 구현하는 측에서는(호출되는 코드에서는) 복잡도가 증가(코딩 양이 증가)해서 구현하기 어려워진다. 관련성이 없는 인터페이스들을 따로 묶어두면 계층이 더 생겨날지는 모르지만, 구현하기가 쉬워지며 역할의 구분과 계층의 구분이 명확하게 나타나게 된다. 그리고, 인터페이스 자체도 역할이 명확해지기 때문에 더 이해하기 쉬운 코드가 된다.

C언어에서 이것을 적용하기 위해서는 인터페이스에 해당하는 함수에 대한 정의가 명확하게 드러나도록 해야한다. 그리고, 그런 함수들을 묶어서 관리해줄 수 있는 방법이 필요하다. 헤더 파일을 이용해서 관련된 함수들을 목적에 맞게 구분해 주고, 각각의 헤더 파일 하나당 구현 파일 하나를 가지는 것도 좋다. 즉, 호출하는 측에서는 자신이 필요한 부분만 정확히 가져다 사용할 수 있으면 된다. 헤더 파일로 묶어 주는 원칙은 공통된 자료구조를 다루는 단위가 될 수 있다. 하지만, 자료구조 자체가 너무 크다면 자료구조내의 관련성을 분석한 후에, 자료구조 자체를 나누고 관련된 함수를 분리된 구현 파일로 나눌 수 있다.

- **DIP(Dependency Inversion Principle)**은 일종의 "의존성 제거(Decoupling)" 방법이다. 즉, 구체적인 것에 의존하지 말고 추상적인 것에 의존하라는 뜻이다. 호출하는 코드는 호출받는 코드의 세밀한 부분(Detail)에 의존하지 말아야 하며, 호출받는 코드도 제공된 인터페이스를 충분히 만족시킬 수 있도록 구현되어야 한다는 뜻이다. 호출하는 코드는 자신이 의존하는 대상을 위해 인터페이스를 정의하고, 이렇게 정의된 인터페이스를 호출받는 코드가 구현하게 된다. 따라서, 기존의 호출하는 코드가 호출받는 코드에 의존성을 가지는 대신에, 호출받는 코드가 호출하는 코드의 인터페이스에 대해서 의존하도록 구현된다는 점에서 "의존성 역전(Dependency Inversion)"이라고 볼 수 있다.

C언어에서 DIP를 구현하기 위해서는 의존성이 생기는 부분에 대해서 인터페이스를 정의한 후, 이를 새로운 계층으로 표현하는 것이 좋다. 그리고, 이를 구현하는 코드에서 정의된 인터페이스를 채워주도록 한다. 새로운 계층으로 표현된 인터페이스가 이미 존재하는 코드와 호환되지 않을 경우에는, 호환을 위한 추가적인 코드가 필요할 수 있다. 호출하는 코드에서는 자신이 필요한 인터페이스에만 의존성을 가지게 되며, 인터페이스 자체는 호출되는 코드에서 구현하기에 인터페이스를 기준으로 두 개의 코드가 계층적으로 나누어지게 된다. 또한, 기존의 이미 있는 코드를 이용해서도 구현할 수 있는 분리된 계층이 존재하기에, 코드의 재사용 가능성도 높아지게 된다. 소프트웨어에서 복잡성을 해결하는 방법은 이처럼 계층적인 구조를 가지는 것이다.

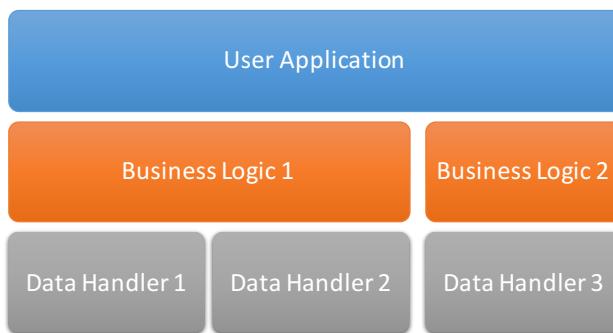
[참고]

C언어를 객체지향 언어처럼 사용하기 위해서 “Go”언어를 배워보는 것도 좋다. “Go”는 C언어를 사용하더라도 어떻게 하면 객체지향 언어처럼 사용할 수 있는지 좋은 힌트를 줄 것이다.

[소프트웨어 아키텍처(Software Architecture)]

소프트웨어의 개발을 건축에 비유하는 경우가 많은데, 건물을 지을 때 필요한 설계도와 모델링 등도 소프트웨어를 개발하는데 있어 필수적인 요소다. 즉, 미리 다양한 생각들을 구성해서 규칙을 만들고, 이를 구현에서 기준으로 활용한다. 만들어야 할 시스템에 대한 명세(Specification)가 명확하게 주어지는 경우는 드물지만, 대략적인 명세를 가지고 설계를 시작하게 해서 구현과 설계를 반복하는 과정에서 더 명확한 명세가 만들어진다. 설계는 구현을 위한 기준을 제공하기에 무턱대고 코딩부터 시작하면 나중에 수정되어야 할 부분들이 늘어나게 되며, 그런 것들이 쌓이게 되면 수정하는 비용이 새로 짜는 것 보다 더 많이 들어갈 수도 있다. 따라서, 생각을 정리해서 코딩을 어떻게 할 것인가를 구현 전에 미리 정하는 것이 좋다.

설계는 시스템을 구성하는 요소를 정하는 일부터 시작한다. 그리고, 그렇게 정해진 요소들간의 관계를 설정한다. 이렇게 만들어진 설계서를 "소프트웨어 아키텍처(Software Architecture)"라고 부르며, 이런 업무를 주로 담당하는 사람을 "아키텍트(Architect)"라고 부른다. 시스템을 구성하는 요소들은 시스템이 구현해야 할 기능들에 의해서 좌우되며, 요소들간의 관계는 비기능적인 것들이 결정한다. 따라서, 기능적인 것과 비기능적인 명세를 확보하는 것은 중요한 일이며, 결과적으로 사용자를 만족시킬 수 있는 소프트웨어를 만드는 가장 핵심적인 활동이라고 할 수 있다.



구조적인 문제를 일으키는 중요 원인은 아키텍처를 만들 때 결정했던 내용들이 디자인이나 구현에서 제대로 반영되지 않는다는 것이다. 아키텍처는 비 기능적인 요구사항을 반영하기 위해서 만들어진 결과물이며, 여러가지 대안 아키텍처 중에서 선택된 이유를 가지고 있어야 한다. 따라서, 그렇게 결정된 사항들을 후속 단계에서 제대로 반영하지 않는다면, 제대로 구현되었다고 볼 수 없다. 좋은 아키텍처란 비 기능적으로 주어진 요구사항을 충실히 만족시킬 수 있는 구조를 말하며, 요구사항과 아키텍처 간에는 양방향 연결 관련(Bidirectional Traceability)을 맺을 수 있어야 한다. 즉, 요구사항의 각 항목에 대해서 아키텍처에 충분히 반영되어야 한다. 후속 단계에서는 만들어진 아키텍처를 통해서 세부적인 사항들을 구체화시키는 과정의 반복이다.

아키텍처는 만들어진 모듈간의 관계를 설명한다. 따라서, 시스템의 통합 테스트에 대한 기준을 제공한다. 어떤 모듈들이 어떻게 결합되어 검증되어야 하는지 표시해주는 것이다. 따라서, 아키텍처는 통합 테스트의 입력으로 사용될 수 있으며, 만들어질 시스템이 유용한지를 미리 알 수 있도록 만든다. 개별 모듈은 문제가 없다고 하더라도, 통합된 모듈은 기능 및 비 기능적인 측면에서 오류를 만들수 있기 때문이다.

[플랫폼(Platform) vs. 프레임워크(Framework)]

플랫폼과 프레임워크라는 두 단어는 일반적으로 흔히 소프트웨어와 관련해서 자주 사용된다. 물론, 그 의미를 정확히 알고 사용하는 경우는 드물지만, 흔히 말하는 흐름(Trend)에 대해서 한마디 거들고자 할 때는 어김없이 등장하는 것도 사실이다. 어떤 용어를 사용할 때는 듣는 사람의 지식 정도에 따라 의미 차이가 발생하겠지만, 그래도 최소한 정확한 상황에서 정확한 어휘를 사용하는 것이 중요하다. 많은 어려운 단어를 사용한다고 해서 더 똑똑한 것도 아니며, 적절한 어휘를 사용해서 듣는 사람의 입장(눈 높이에서)을 충분히 고려해서 이야기 하는 것이 최선일 것이다.

플랫폼은 어떤 사건(혹은 일)이 일어나는 토대를 제공하는 곳이다. 기차를 타기 위해서 사람들이 대기하는 장소나 연설자가 올라서서 이야기하는 곳을 플랫폼이라고 하며, 소프트웨어 개발에서는 응용 프로그램이 동작하는 환경을 제공하는 것이 플랫폼이다. 따라서, 플랫폼은 일의 주체적인 입장이 아니라 요청을 받아 일을 처리하는 곳이며, 처리된 결과는 다시 원하는 응용 프로그램에게 전달된다. 종종 플랫폼과 운영체제(OS: Operating System)을 혼동해서 사용하는 경우도 있지만, 운영체제 자체는 하드웨어 자원의 관리를 맡고 있는 소프트웨어일 뿐이며, 확장된 기능과 데이터의 해석 등은 어플리케이션이 담당한다.



프레임워크는 그 자체가 완벽히 동작할 수 있는 응용 프로그램이다. 즉, 따로 사용자가 작성한 코드가 없더라도 동작하는데 문제가 없다. 물론, 프레임워크가 동작하기 위한 환경은 운영체제나 플랫폼 등에서 제공받아야 하지만, 그렇다고 사용자가 코딩 할 필요는 없다. 이렇게 만들기 위해서는 프레임워크 자체가 기본적(Default)으로 실행해야 하는 일을 가지고 있다는 것을 알 수 있다. 또한, 사용자가 작성하는 코드의 시작점이 없기에, 제어도 프레임워크 자체가 전담(Inversion of Control)한다고 볼 수 있다. 사용자는 기본적으로 프레임워크가 정의한 동작을 재정의(Override) 하는 형태로 코드를 확장할 수 있으며(Extensible), 프레임워크 코드 자체의 변경은 용납하지 않는다(Non-modifiable). 따라서, 프레임워크는 쉽게 사용자가 코딩할 수 있는 방법을 제공하지만 의존성도 높아질 가능성을 가지고 있다.

계층적인 관점에서 본다면, 응용 프로그램의 개발에는 프레임워크를 사용하는 것이 개발 효율이 높다. 하지만, 기본 기능의 확장이나 변경을 통해서만 개발해야 하기 때문에, 플랫폼에 비해서 상대적으로 자유도는 낮다. 플랫폼은 개발의 토대만을 제공하기에, 응용 프로그램을 효과적으로 개발하는데는 한계를 가지지만, 프레임워크에 비해서 확장이나 변경이 용이하다. 따라서, 계층적으로 구성하면, 최상위에는 응용프로그램을 구축하는 프레임워크가, 그 아래에 플래폼이, 그리고 플랫폼의 일부에 운영체제와 같은 것이 오게된다. 프레임워크를 사용하더라도 코드의 의존성을 줄이고자 한다면, 프레임워크에 의존하는 부분과 로직을 분리하는 방법을 사용해야 한다. 마치 UI와 비지니스 로직을 분리해서 관리하듯이, 프레임워크에 의존적인 부분과 그렇지 않은 부분을 분리해서 코딩해야 재활용 가능성을 조금이라도 더 높여줄 수 있다.

[좋은 이름 만들기]

코딩은 이름 붙이기의 연속일지도 모른다. 좋은 이름은 코드를 읽기 쉽게 만들어 주어야 하며, 읽는 사람이 오해할 가능성이 줄여 주어야 한다. 하지만, 정작 이름을 붙이려고 하는 순간 어떤 이름을 붙여야 할지 망설여지는 경우가 많다. 여기서는 이름 붙이기에 대해서 몇 가지 필요한 것들을 이야기해 보겠다.

사물을 이름붙이면 사람은 추상적인 사고를 할 수 있게된다. 예를 들어, 집에 있는 강아지를 1번 2번이라고 부르는 것보다는 "복순이", "순돌이"와 같이 이름 붙이는 것이 더 정겹게 들리고 구분하기도 쉽다. 이름은 사물을 생각할 때, 그것이 가지는 의미를 적절히 표현할 수 있어야 좋은 이름이라고 할 수 있다. 즉, 전혀 관련없는 이름을 붙인다면, 처음 보는 사람의 입장에서는 그것이 무엇을 의미하는지 고민해야 한다. 이름은 이해를 돋기 위해서 역할을 짐작할 수 있는 것이 좋다.

1. 함수의 이름은 "동사+명사"로 만드는 것이 좋다.

; 함수의 이름에 모듈의 이름을 약자로 명시하는 경우가 있지만, 반드시 그럴 필요는 없다. 함수의 이름은 한 가지 일을 하는 것처럼 보여야 한다. 그리고, 실제 구현도 한가지 일만 해야한다.

함수가 좋은 이름을 가지기 위해서는 함수가 하는 일이 명확해야 한다는 것을 알 수 있다. 여러가지 일을 하면 "common", "util", "xxx_and_yyy"와 같은 이름이 만들어질 것이다. 이것은 좋은 이름이 아니다. 만약, 조건문과 같은 곳에서 사용되는 "조건"을 함수화했다면, 질문과 같은 형식도 좋다. "isXXX()"와 같이 물어보는 이름은 질문의 답을 구하게 된다는 것을 쉽게 예상할 수 있기 때문이다.

2. 변수의 이름은 묘사적으로 만들고, 결과를 저장하는 경우에는 타겟이 되는 명사가 좋다.

; 변수는 직접적인 대상으로 사용 되기에 명사를 위주로 적는다. 추상적인 명사를 사용할 수도 있으며, 가급적이면 딱 하나로 지정할 수 있어야 한다. 단수나 복수를 나타내는 것도 좋지만, 혼돈의 우려가 있을 경우에는 단수명으로 통일한다.

자주 사용하는 일반화된 관행은 따로 이름을 붙일 필요는 없다. 예를 들어, "for()"나 배열의 인덱스로 사용하는 값들은, 그냥 "i, j, k"와 같은 이름을 사용해도 혼돈의 이유가 없다. 하지만, 만약 그런 인덱스가 특별한 의미로 사용된다면 이름을 가지는 편이 코드를 이해하기 쉽게 만들 것이다. 변수는 주로 사용되는 범위에 따라 이름의 길이를 길게 하는 것이 좋다. 전역적으로 사용되는 변수라면 명확한 역할을 묘사적으로 서술해야 한다.

3. 일반 상수도 모두 이름을 붙여 준다.

; 대문자로 표시하는 것이 변수와 구별할 수 있어서 좋다. 주로 군집의 형태를 나타내는 경우에는 "enum"을 사용해서 묶어준다. "enum"에는 동일한 종류의 것들만 묶는 것이 원칙이다. 만약, 다른 종류를 묶어야 한다면, 차라리 "#define"으로 처리하는 것이 좋다. "enum"으로 묶어 줄 경우에는 "enum"의 이름을 상수의 이름에 붙일 필요가 없다. 반복할 필요가 없는 것이다.

코드를 보게될 때 가장 힘든 부분은 아무런 설명이 없는 상수다. 간혹 옆에 주석으로 의미를 적어놓기도 하지만, 차라리 이름을 붙여주는게 이해를 돋는다. 특히, 상수는 그냥 사용하면 코드 수정을 방해하는 주요 요인으로 작용하기에, 모두 제거하는 것이 좋다. 상수 옆에 주석을 달아서 이해를 도우려고 하는 경우가 있지만, 그것보다는 상수 자체에 의미있는 이름을 주는 것이 더 좋은 선택이다.

4. 조건문을 차지하는 함수는 질문 형태를 가져야 한다.

; 질문은 원하는 것을 검증하는데 필요한 긍정적인 것이 좋다. 예를 들어, 빈 것을 알고 싶다면 "isEmpty()"가 좋지만, 비지 않았는지를 확인하고 싶다면 "isOccupied()"로 바꿔야 한다. 즉, 물어보는 것이 긍정적으로 받아들일 수 있으면 읽는 것도 편해진다. 사람은 긍정적인 것을 더 쉽다고 생각한다. "!"(Not)이 들어간 조건문은 이해하기 힘들다. 따라서, 읽는 사람의 입장을 고려한다면, 될 수 있으면 긍정적인 표현을 사용하도록 바꾼다. 물론, 그 값은 "true, false"를 두 가지 다 가져도 상관없다.

조건이 하나인 경우에도 될 수 있으면 긍정적인 표현을 하도록 한다. 조건이 2개인 경우에는 드모르강의 법칙과 같은 것을 사용해서 긍정적인 조건들로 변경할 수 있을 것이다. 조건이 3개 이상이 될 경우에는 함수를 만드는 것을 고려해 보아야 한다. 조건이 늘어나면 이해하는 것도 쉽지 않으며, 테스트 해야할 경우의 수도 늘어난다. 따라서, 이와 같은 경우에는 조건이 의미하는 바를 설명할 수 있는 함수를 만들어 대체하는 것이 좋을 것이다.

5. 파일의 이름은 파일이 하는 역할을 설명할 수 있어야 한다.

; 파일의 이름을 정했다면, 내부에는 그 이름과 관련된 함수나 자료구조만 포함하는지 확인해야 한다. 만약 그렇지 않다면, 파일의 이름을 바꾸거나, 새로운 파일을 만들어서 일치하지 않는 함수를 옮겨주어야 한다. 파일은 지나치게 길어선 안된다. 문제가 많이 발생하는 파일들은 대부분 코드가 길게 작성된 경우가 많다. 함수가 길어도 문제가 많이 발생하지만, 긴 파일들도 문제가 많이 발생한다. 따라서, 작은 파일

들로 나누는 것이 좋다.

파일의 길이가 특정 크기로 정해진 것은 아니지만, 하나의 함수를 대략 50(그보다 적어도 되지만)라인 정도라고 가정하면, 그런 함수들이 20개 정도가 들어갈 수 있는 길이인 1,000라인 이상은 되지 않는 것이 좋다. 길이가 긴 파일은 내부에 다양한 함수들이 섞여 있을 가능성이 높으며, 파일 이름이 너무 일반적일 가능성이 있다. 즉, 파일 이름으로 그 역할을 추정할 수 없다는 뜻이다. 따라서, 파일이 하는 역할을 명확하게 만들기 위해서, 서로 관련이 있는 함수와 자료구조 들을 분리해서 파일로 묶어 주어야 할 것이다. 각각의 이름은 그 역할을 추측하도록 붙이면 된다.

6. 디렉토리의 이름은 파일들의 이름을 유추할 수 있는 이름이어야 한다.

; 디렉토리속에 어떤 파일들이 있는지 디렉토리 이름을 통해서 파악할 수 있어야 한다. 따라서, 전혀 관련이 없는 파일이 같은 디렉토리에 있지 않아야 한다. 파일들이 섞여 있다면, 디렉토리를 새로 만들어서 관련이 적은 파일들을 디렉토리 별로 모아주어야 한다. 디렉토리들 간에도 계층은 존재한다. 물리적인 계층이 논리적인 계층과 일치하면 좋겠지만, 그렇게 만드는 경우에는 디렉토리가 깊게 계층적으로 만들 어진다. 따라서, 논리적으로 하나의 모듈이 하나의 디렉토리로 구성하는 것이 좋다.

당연히 디렉토리들 간에는 의존관계를 명확히 하기 위해서 외부의 요청을 받는 역할을 하는 파일들과, 외부로 요청을 담당하는 파일들을 유지하는 것이 좋다. 즉, 외부 사용자가 알아야 하는 정보와 외부에 의존적인 정보를 분리해서 관리하면, 사용하는 측도 편해지고 디렉토리에 들어있는 코드들을 포팅 하기도 편해진다. 아키텍처 설계서를 가지고 있다면, 그 설계서에 맞춰서 디렉토리를 구성할 수도 있을 것이다. 가능한 작성된 문서와 유사한 형태로 디렉토리 구조를 가져가면, 전체적인 시스템 설계와 코딩을 비교하 기 쉬우며, 어디서 수정해야 할지 빨리 파악할 수 있다.

코드의 불록 들도 이름을 주어서 함수로 만들면 여러가지로 도움이 될 수 있다. 즉, 작은 코드 조각들 이라도 함수로 만들어서 사용하게 되면, 나중에 그 코드를 재활용할 가능성이 높다. 그리고, 코드를 읽는 입장에서는 상세한 구현을 보지 않더라도, 함수 이름만 가지고도 무슨 일을 하는지 파악할 수 있게된다. 코드는 결국 사람이 읽기 위해서 만들어지기에, 좋은 이름을 역할과 관련 맺어주는 것이 읽는 사람에게는 큰 도움이 될 것이다.

[함수(Function), 프로그래밍의 최소단위]

C언어는 독립적으로 실행할 수 있는 가장 작은 단위(Unit)로 함수(Function)를 사용한다. 참고로 C++ 과 같은 객체지향 언어에서는 가장 작은 단위로 클래스(Class)를 가진다. 따라서, 하나의 함수를 잘 만드는 것은 집을 짓는데 비유하면, 좋은 벽돌을 만드는 것과 같다고 볼 수 있다. 좋은 재료를 가지고 있어야 좋은 건축물을 지을 수 있듯이(물론, 좋은 설계도 중요하다), C언어에서는 좋은 함수를 만드는데 노력을 집중해야 한다. 그럼 좋은 함수란 무엇일까? 좋은 함수가 가져야하는 특징은 아래와 같이 요약할 수 있다.

01. 함수는 짧을 수록 이해하기 쉽다.

; 함수의 길이가 길어지면, 이해가 어려운 수준에 도달하게 된다. 대략, 한 화면을 기준으로 크기를 정하면 일반적으로 별 문제가 없다. 최근에는 해상도가 높은 모니터가 많은데, 이때는 한 화면에 2개의 함수 가 들어올 수 있는 길이 정도가 적당할 것이다.

02. 테스트를 위해서는 출력이 있는 것이 좋다. 즉, 실패했을 때와 성공했을 때를 구분할 수 있어야 한다.

; 리턴(Return) 값이 없는 함수도 좋지만, 함수의 실행 결과를 알려줄 수 있으면 실패 여부를 알 수 있다. 디버깅 목적으로 한정해서 사용하는 경우에는 배포에서 제외할 방법을 만들어두면 된다.

03. 될 수 있으면 다른 함수나 모듈에 의존하지 않아야 한다. 즉, 다른 함수나 모듈을 호출하는 것이 적을 수록 좋다.

; 의존이 많은 함수는 의존하고 있는 함수나 모듈이 변경될 때 영향을 받게 된다. 물론, 그렇다고 전혀 의존성이 없는 것도 불가능 하기에, 될 수 있으면 의존성을 줄이도록 만들어야 한다. 같은 파일에 정의된 함수들을 사용하는 것은 그나마 의존성이 약하지만, 다른 모듈에 정의된 함수나 자료구조를 이용하는 것은 의존성을 높인다.

04. 함수의 이름은 함수가 하는 일을 정확히 설명할 수 있어야 한다.

; 함수 이름이 내부에 구현된 코드가 하는 일을 명확히 설명해야 한다. 함수의 이름만 보고도 그 함수가 하는 일을 파악할 수 있어야 한다. 만약, 이름과 다른 일을 하는 함수가 있다면 잘못 구현된 것이다.

05. 함수는 한가지 일만 해야 한다.

; 여러가지 일을 하는 함수는 이름에서 이미 “일반적인” 의미를 가진다. 혹은, “그리고(And)”와 같은 이름을 가질 수도 있다. 함수는 잘 정의된 한가지 일을 간결하게 처리해야 한다.

06. 함수는 입력 인수(Parameter)가 없을수록 좋다. 가장 좋은 것은 아예 인수가 없는 함수다.

; 함수의 인수가 많을수록 복잡한 일을 하고 있을 가능성이 높다. 또한, 같이 사용 되는 데이터들이 자료구조의 형태로 만들어지지 않았을 수도 있다. 함수가 입력 인수가 많이 필요할 경우, 구조체(Structure)를 이용해서 인수를 묶어서 관리하는 것이 좋다. 함수의 입력 인수 개수를 줄이기 위해서 구조체를 정의하고, 관련 함수를 같이 묶어서 관리해야 한다. 자료구조와 그것을 다루는 함수는 항상 같이 움직여야 한다.

07. 함수의 내부에서 사용하는 자료구조는 외부로 될 수 있으면 보여주지 않아야 한다.

; 자료구조를 외부로 노출하는 것은 직접적인 접근을 허락한다는 말이다. 이것은 자신이 처리를 담당하는 것이 아니라, 모듈들이 자유롭게 사용하도록 허락하는 것이다. 따라서, 자료구조의 변경이 언제 어떻게 발생할지 알 수 없게되며, 자료구조 자체의 변경이 다른 코드에 주는 영향도 크게 만든다. 함수는 가능한 사용하는 측에서 내부 구조를 모르게 만들어야 한다. 함수의 외부로 자료구조를 보여주어야 할 경우에는, 자료구조를 생성하는 함수를 추가로 구현해서 내부구조를 숨긴다. 향후에 이렇게 생성된 자료구조를 필요한 함수들의 인수로 사용한다. 자료구조를 관련된 함수들이 같이 알고 있어야 하는 경우, 사용하는 측에 감추기 위해서 이와 같은 방법을 사용할 수 있다. 일종의 “핸들(Handle)”개념을 사용하는 방법이다.

08. 함수의 입력 인수들은 값의 유효성 범위가 검증될 수 있어야 한다.

; 함수는 잘못 사용될 가능성이 항상 있다. 함수를 호출하는 측에서 오류를 제공할 가능성이 있기에, 입력되는 값에 대해서 유효한 범위에 있는지 확인할 수 있어야 한다. 충분한 테스트를 했다고 한다면, 배포 시에는 제외될 수 있다.

09. 각각의 함수마다 테스트 할 수 있는 프로그램을 충분히 만든다.

; 함수는 단위 테스트의 기본 단위다. 부품의 완성도를 높이는 것이 전체 제품의 완성도도 높인다. 따라서, 최소한 외부로 노출되는 함수 들에 대해서는 단위 테스트를 만들어야 할 것이다. 내부에서 사용 되는 함수들은 외부로 노출된 함수들이 호출해 줄 것이다. 외부에 노출되는 함수의 단위 테스트는 함수 자체의 사용법을 알려주는 문서와 같은 역할을 할 수도 있다.

10. 각 함수들은 중복된 코드가 없어야 한다. 있을 경우에는 이를 뽑아내서 다른 함수로 만든다.

; 모든 중복은 오류의 온상이 될 수 있다. 코드의 중복은 흔히 발생하며, 개발자들은 무의식적으로 중복을 만든다. 중복된 코드는 함수화 시키는 것이 해결방법이다.

추가적으로 함수의 리턴문은 가능한 적을수록 좋지만, 일반적으로 처리가 끝났으면 빨리 복귀하는 것이 좋다. 물론, 이때는 제어의 이동이 발생하기에 할당된 자원의 처리를 반드시 완료한 후에 해야한다는 것을 기억한다.

이곳에서 본 것들이 좋은 함수를 만들기 위한 모든 방법은 아니다. 개발자마다 자신의 경험이 틀리기 때문에, 다양한 팁(Tip)들이 있을 수 있다 하지만, 대체적으로 위에서 본 것들은 기본적으로 좋은 코드를 만들기 위해서 가져야 할 것들이기에 기억하는 것이 좋을 것이다.

[연산자의 우선 순위를 걱정하나?]

솔직히 연산자의 우선 순위는 크게 기억할 필요가 없다. 왜냐면 "("와 같은 것을 충분히 사용해서 구분해 주면, 오류가 날 가능성이 줄어들기 때문이다. 하지만, 기본적인 우선 순위정도는 기억할 필요가 있으며, 일반적인 사칙 연산 외에 조합되는 연산 들에 대해서는 "("를 충분히 사용해 우선 순위를 명확히 보이도록 한다. 가장 쉽게 틀리는 부분은 "*"를 사용해서 포인터(pointer)로 이용하는 부분인데, 이때도 포인터라는 것을 명시적으로 나타내주면 문제될 것은 없다. 읽는 사람이 생각을 하게 만들지 않는 코딩이 최선이다. 읽는 것에 막힘이 없다면 좋은 코딩인 것이다.

```
tax = ( income x tax_rate + unpaid_tax ) / discount_rate;
->
tax = (( income x tax_rate ) + unpaid_tax ) / discount_rate;
```

연산자들과 "("를 같이 섞어서 사용할 때 가장 틀리기 쉬운 곳은 매크로(Macro)를 정의하는 경우다. 매크로는 말 그대로 1:1로 대체(Replace)가 발생하기에, 제대로 "("를 사용하지 않을 경우에는 엉뚱한 값을 만들어 낸다. 그리고, 매크로는 부수효과(Side Effect)을 가질 가능성이 높기에 복잡한 매크로를 사용할 경우에는 차라리 함수(Inline함수와 같은)를 만들어서 사용하는 것이, 버그도 줄이고 코드의 이해도 쉽게 만들 수 있는 방법이다.

[포인터(Pointer)에 대해서]

C언어를 배우는데 있어서 가장 큰 장애물은, 많은 사람들이 지적했듯이 포인터(Pointer)의 사용에 관련되어 있다. 포인터는 주소(Address)를 적어두는 변수로 수시로 값을 변경할 수 있다(물론, 상수 포인터로 선언된 경우에는 변경이 불가능하며, 선언 시에 초기화도 해주어야 한다.).

포인터는 특정 메모리를 가리키는 역할을 하기에, 포인터의 값을 즉각적으로 사용하지 않고, 그 값을 다시 이용해서 원하는 주소를 찾게 된다. 찾아간 주소에는 데이터가 들어있을 수도 있고, 코드가 들어갈 수도 있다. 즉, 포인터는 자신이 가리키는 값에 대해서는 사용자가 쓸 수 있도록 형(Type)을 지정해 주어야 한다는 말이다. 어떤 내용을 가르키는지를 알아야지 나중에 컴파일러가 실제 동작하는 코드를 만들 때 정확한 형을 검사할 수 있기 때문이다.

포인터는 다양한 유용성이 있지만, 하드웨어를 제어하는 코드에서 사용될 수 있다. 즉, 특정 주소(대부분의 경우, 논리적인 주소보다는 물리적인 주소를 사용한다. 물론, 운영체제와 같은 것이 있다면, 가상의 주소를 물리적인 주소로 변환해 준다.)에 대해서 데이터를 읽거나 쓸 경우에 포인터를 사용할 수 있다. 예를 들어, 0xFFFF0000이라는 주소에 데이터를 쓰는 것을 아래와 같이 할 수 있다.

```
const int *buffer = (void *)0xffff0000; /* 포인터가 가리키는 곳을 변경할 수 없음 */
int * const pointer = (void *)0xffff0000; /* 포인터 자체가 변경 안됨 */
```

```
*buffer = 0x0000ABCD; /* 값을 쓰려고 하면 컴파일 안됨 */
buffer = (void *)0x0000ABCD; /* 포인터 값 자체는 변경할 수 있다. */
*pointer = 0x0000ABCD; /* 포인터가 가르키는 곳에 값을 쓸 수 있음 */
pointer = (void *)0x1111FFFF; /* 포인터 값 자체를 변경할 수 없다. */
```

포인터는 자신의 값을 변경할 수 있는지, 혹은 자신이 가리키고 있는 곳의 값을 변경할 수 있는지 정확히 표현해 주어야 한다. 만약, 읽기 전용인 곳에 대해서 사용한다면, 앞에서 “buffer”와 같은 것을 선언하는

것처럼 해 주어야 할 것이다. 이때, “buffer” 자체는 변경될 수 있다는 점을 주의해야 한다. “pointer”的 경우에는 포인터 자체가 변경이 불가능하기에, 하나의 주소 값만 가지게 된다.

[재귀(Recursive) 호출]

문제를 구현하는 방법은 다양하게 있을 수 있다. 하지만, 최적의 해를 구하는 것은 생각보다 어렵다. 재귀적 호출을 통해서 구현할 수 있는 것들이 다양하게 있겠지만, 이를 실제로 구현해서 성능을 높이는 것은 쉽지 않다. 재귀적 호출에 들어가는 비용이 상대적으로 비싸다. 즉, 중간 결과를 이용할 수 있도록 구현되어야 호출의 깊이가 짧아질 수 있다(이를 다른 말로 동적 프로그래밍(Dynamic Programming)이라고도 함). 예를 들어, 피보나치(Fibonacci) 수열을 생각해 보자.

1, 1, 2, 3, 5, 8, 13, 21...

즉, 첫 두 값을 제외하고는 이전의 2개의 값의 합으로 다음 수가 정해진다. 이를 구현하는 방법은 함수가 자신을 재귀적으로(Recursive) 호출하는 방법을 사용할 수 있다.

$$F(2) = F(0) + F(1)$$

$$F(3) = F(1) + F(2)$$

$$F(4) = F(2) + F(3)$$

...

따라서, 이를 구현하는 코드는 아래와 같다.

```
int fibonacci( int nth ) {
    if( ( nth == 0 ) || ( nth == 1 ) ) {
        return 1;
    }
    return ( fibonacci( nth - 1 ) + fibonacci( nth - 2 ) );
}
```

`fibonacci()`함수는 구현은 짧고 이해하기 쉽게 보인다. 하지만, 이를 실행하면 이야기는 달라진다. 예를 들어, 200번째 피보나치 수를 구하기 위해서는 무수한 중간 결과물에 대한 계산이 중복적으로 발생하고, 이 때문에 필요한 스택의 크기도 커진다. 따라서, 이해하기 쉬운 구현이라고 해도 좋은 성능을 보장하기에는 부족한 부분이 있다. 하지만, 이것은 이해하기 쉬운 코드가 전부 성능이 낮다는 것을 보여주는 예로는 부적합하다.

앞에서 이미 이야기 했듯이 문제의 해결방법은 다양하게 있을 수 있다. 앞에서 `fibonacci()`함수는 구현의 한 예일 뿐이다. 즉, 방법을 달리 구현해서 성능을 높이는 것이 가능하며, 이때도 역시 이해하기 쉬운 코드를 만드는 것이 가능하기 때문이다. 여기서 구현한 `fibonacci()`함수는 단지 하나의 해법에 불가하며, 수학적인 정의를 그대로 코드로 구현해서 생기는 문제를 보여주기 위함이다. 따라서, 성능상의 문제를 발견했고, 그 문제가 어디서 생기는 것인지를 이미 안다면 개선할 수 있는 방법을 찾으면 된다.

```
#include <stdio.h>
#include <stdlib.h>

int fibonacci(int nth) {
    if ((nth == 0) || (nth == 1)) {
        return 1;
    }
    return fibonacci(nth - 1) + fibonacci(nth - 2);
}
```

```

int main(void) {
    int nth = 50;

    puts("This is Fibonacci Number Generation Project!!!\n");
    printf("The %dth Fibonacci Number is : %d\n", nth, fibonacci(nth));

    return EXIT_SUCCESS;
}

```

"nth"가 50만 되어도 실행되는데 상당한 시간이 걸린다는 것을 알 수 있을 것이다. 이것을 조금 바꿔서 실행 시간을 줄여야 사용자에게 의미있는 결과를 제한된 시간 내에 제시할 수 있다. 변경은 중간 계산값을 저장하는 방식이면 충분하다.

```

#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE_OF_FIBONACCI 50
long storeFibonacciValue[MAX_SIZE_OF_FIBONACCI + 1] = { 0 };

long fibonacci(long nth) {
    if ((nth == 0) || (nth == 1)) {
        storeFibonacciValue[nth] = 1;
        return 1;
    }

    if ((storeFibonacciValue[nth - 1] != 0)
        && (storeFibonacciValue[nth - 2] != 0)) {
        storeFibonacciValue[nth] = storeFibonacciValue[nth - 1]
            + storeFibonacciValue[nth - 2];
    } else {
        storeFibonacciValue[nth] = fibonacci(nth - 1) + fibonacci(nth - 2);
    }
    return storeFibonacciValue[nth];
}

int main(void) {
    int i;
    long nth = 50;

    puts("This is Fibonacci Number Generation Project!!!\n");
    printf("The %ldth Fibonacci Number is : %ld\n", nth, fibonacci(nth));

    for (i = 0; i < MAX_SIZE_OF_FIBONACCI; i++) {
        printf(" %ld", storeFibonacciValue[i]);
    }
    puts("\n");

    return EXIT_SUCCESS;
}

```

위의 코드는 중간 결과값을 저장해서 이를 이용할 수 있도록 만든 것이다. 다소 지저분해 보이기는 하지만(이는 나중에 계선해야 한다.), 앞에서 만든 코드 보다는 더 빨리 동작할 것이다. 물론, 저장하는 값의 범위가 넘어가서 마지막에 속하는 값들은 음수로 나올 수도 있다. 중요한 점은 재귀적인 호출을 이용하는 것이 도움이 될 경우도 있지만, 실제로 이를 구현할 때는 성능과 확장성을 같이 고려해야 한다는 점이다. 위의 코드는 계산 범위를 50번째 까지로 제한했지만, 필요한 경우 메모리를 할당하는 방식으로 변경해도 상관없다.

피보나치 수열 프로그램을 개발 할 때, 몇가지 테스트 케이스를 나름 만들어서 시도했다. 즉, 조건에 명시된 0번째, 1번째에 대한 테스트와 3번째에 해당하는 값을 "nth"에 넣어서 코드에 문제가 없는지 확인했다, 그리고, 다시 10번째, 50번째로 테스트를 했다. 나름 경계조건일 수 있는 값들을 넣어서 테스트를 실행한 것이다.

이런 작은 함수를 구현하는데도 여러번의 테스트가 필요하다는 것이다. 필요하다면 테스트를 위한 프레임워크를 사용하는 것도 좋다. 여기서 중요한 점은 항상 테스트를 할 수 있는 도구를 옆에 두고, 코드의 작은 변화를 감지할 수 있도록 자동화된 테스트를 만들라는 것이다. 그것이 굳이 TDD(Test Driven Development)가 아니라도 상관없다. 실해할 수 있는 테스트만이 자신이 개발한 코드에 문제가 없음을 검증하는 방법이다.

```
#include <stdio.h>
#include <stdlib.h>

unsigned long long fibonacci(int nth) {
    int i;

    unsigned long long first = 1;
    unsigned long long second = 1;
    unsigned long long fiboValue = 0;

    if (nth < 0) {
        /* We cannot calculate the fibonacci number in less than 0th. */
        return 0;
    }

    if ((nth == 0) || (nth == 1)) {
        return 1;
    }

    for (i = 2; i <= nth; i++) {
        fiboValue = first + second;
        first = second;
        second = fiboValue;
    }

    return fiboValue;
}

int main(void) {
    int i;
    int nth = 60;

    puts("New Fibonacci Number Generator");
}
```

```

for (i = 0; i <= nth; i++) {
    printf("The fibonacci value of %dth is %llu.\n", i, fibonacci(i));
}

return EXIT_SUCCESS;
}

```

위의 코드는 앞에서 구현한 코드를 조금 더 수정한 것이다. 이 코드는 중간값의 저장을 하지 않고, N번째의 피보나치 수를 계산하기 위해서 바로 전의 두개의 값("first", "second")만을 보관해서 계산한다. 중간값을 저장하기 위해서 필요했던 배열을 제거한 코드다. 이처럼 재귀적인 호출로 구현된 코드를 루프(Loop)를 통해서 구현할 수 있으며, 계산 성능도 개선할 수 있음을 보여준다. 여기서도 마찬가지로 0, 1, 2, 3, 10, 20, 40, 50으로 "nth"를 증가시키면서 제대로 된 값이 나오는지를 확인했다. 경계 조건이 될 수 있는 값을 가지고 테스트를 통과하는지 판단해야 한다. 이를 자동화 시키는 것은 그렇게 어려운 일은 아닐 것이다.

추가적으로 큰 값을 저장하기 위해서 변수의 타입도 "unsigned long long"으로 변경했다. 50번째의 피보나치 수를 구할 때 발생했던, 오버플로우(Overflow)문제도 사라졌으며, 구할 수 있는 범위도 더 확장되었다(그렇다고 무한정 확장할 수는 없으며, 64bit CPU에서 92번째까지 구할 수 있다.). 이 코드를 Eclipse에서 컴파일하면, "printf()"의 포맷(Format)에 이상이 있다는 "warning"메시지가 보일지 모르지만, 실행하는데는 아무런 문제가 없다. 그리고, 임시값을 "static"변수에 저장하는 방식은 멀티쓰레드(Multi-Thread)를 이용해서 프로그램 할 경우에 문제가 발생할 수 있기에, 가급적이면 하나의 쓰레드만 접근하도록 만들어야 했다. 하지만, 이번에 구현된 것은 "static" 변수가 없으며, 지역 변수만을 사용해서 구현하기에 각 쓰레드별 스택(Stack)영역만을 사용해서 멀티쓰레드에서 호출될 경우에도 문제가 없을 것이다.

[전역변수의 사용]

전역변수는 가장 쉽게 데이터를 읽고 쓸 수 있는 방법이다. 전역변수는 프로그램의 어느 부분에서도 접근(Access)할 수 있으며 읽고 쓰는 것이 가능하다. 따라서, 전역변수는 프로그램의 상태를 저장하거나 얻어낼 수 있는 방법으로 선호되며, 시간이 흘러갈수록 점점 더 많은 전역변수를 사용하는 경향이 나타나게 된다. 이미 많이 들었겠지만, 전역변수는 원인을 밝히기 어려운 버그를 만들어 낼 가능성이 높다. 또한, 코드의 수정이 시스템 전체에 영향을 줄 수 있기에, 버그 발생 가능성 자체도 높다.

누구나 아무 곳에서 사용할 수 있다는 말은 아무도 제대로 관리하지 않는다는 말이며, 이런 이유로 인해서 잘못된 값을 읽거나 쓸 가능성이 높아지게 되는 것이다. 모듈간의 의존성(Dependency)은 전역변수를 공유하게 되면 커지기 때문에, 분리가 거의 불가능한 코드를 만들 가능성이 높다. 의존성이란 특정 모듈이 어느 모듈의 변경에 얼마나 영향을 받는가를 나타내며, 다른 모듈에 선언된 전역변수를 직접 접근하는 것은, 해당하는 자료구조가 변경될 때 영향을 받을 가능성을 높이게 된다. 따라서, 이런 전역변수가 많다는 말은 그 만큼 코드를 바꾸기가 어렵다는 말과 동일하게 받아들여야 한다.

전역변수를 없애는 방법은 전역변수를 접근하는 전용 메커니즘을 만들어 넣는 방법이다. 즉, 직접적으로 변수를 접근하는 대신에 간접적으로 접근할 수 있도록 만드는 것이다. 물론, 이때 발생하는 오버헤드(Overhead)는 함수의 호출이나 매크로(Macro)의 정의 등과 같은 것이 있을 수 있으며, 코드의 증가나 실행 성능의 하락이 발생할 수 있다. 하지만, 이런 것들은 사실 미미한 부분이며, 전역변수의 사용을 줄여서 얻는 효과 보다는 작다. 따라서, 직접적인 전역변수에 대한 사용을 가능한 줄이는 것이 더 나은 선택이다.

위와 같이 파일에 선언된 변수는 파일의 외부에서는 접근하지 못한다. 즉, 이 변수를 접근하기 위해서는 특정한 방법을 만들어야 할 것이다. 예를 들어, 앞에서 정의된 변수에 대한 접근자(Accessor)는 다음과 같이 만들 수 있다.

```
static int globalVariable;

static inline void setGlobalVariable(int setValue) { /* 외부에 보여주려면 "static inline"을 제외 */
    globalVariable = setValue;
}

static inline int getGlobalVariable(void) {           /* 외부에 보여주려면 "static inline"을 제외 */
    return globalVariable;
}
```

이렇게 만들어진 함수들은 나중에 전역변수를 접근하는 곳 들을 대체하게 되며, 필요하다면 디버깅 시에 어디서 호출되는지를 파악하기 위한 디버그 메시지를 생성하는데도 사용할 수 있다. `inline`으로 선언한 이유는 이것이 사실상 "Leaf Function"에 해당하기 때문에, 컴파일러에게 함수 호출과 같은 오버헤드를 없앨 수 있다는 것을 알리기 위해서이다. 즉, 다른 함수를 다시 호출하지 않는다면, 레지스터의 내용을 저장하고 복구하는 과정을 생략할 수 있으며, 마치 함수의 호출이 아닌 코드의 대체(Replacement)로 삽입될 수 있도록 만들 수 있다.

전역변수들은 컴파일러가 제대로 최적화를 하는데 방해가 되기도 한다. 즉, 전역변수는 어디서 언제 바뀌게 될지 알 수 없기에, 컴파일러는 매번 메모리에서 값을 불러 들이도록 만든다. 이는 레지스터에 저장된 값을 사용하는 것보다 CPU 사이클(Cycle)이 더 많이 필요하기에, 최적화(컴파일러의 최적화) 시키더라도 성능 개선이 되지 않는다. 만약, 전역변수에 접근하는 연산이 자주 실행되는 루프(Loop)에 있게 되면, CPU 사이클의 상당 부분을 전역변수를 접근하는데 사용하게 된다. 따라서, 전역변수는 단순히 버그 발생 뿐만이 아니라, 프로그램의 성능저하에도 영향을 준다.

가장 좋은 것은 전역변수를 전혀 사용하지 않는 것이다. 물론, 이것이 불가능하다고 생각할 수 있겠으나, 사실 전역변수를 사용하지 않는 코딩은 가능하다. 자료구조를 될 수 있으면 숨기는 것이, 해당 자료구조를 보이도록 만드는 것보다 버그가 적은 것은 당연하다. 즉, 자료구조에 대한 의존성을 가진 코드를 만들지 않게 되기 때문에, 내부적인 구조 변경이 외부에 영향을 줄 가능성성이 줄어들기 때문이다. 변화에 민감한 코드를 짜는 것은 항상 버그를 양산한다. 성능 때문에 무작정 전역변수를 사용하는 것은, 이후에 길고 지루한 디버그 과정과 최적화를 생각한다면 어리석은 선택일 뿐이다.

[문법보다는 눈에 잘보이는 코드 작성하기]

C언어는 다른 언어에 비해서 제약을 적게 두고 있다. 즉, 프로그래머에게 많은 책임을 맡기고 있다. 이런 이유 때문에 의도하지 않은 오류들이 발생할 수 있으며, 프로그래머가 이러한 오류를 찾아내는 것은 어렵고 힘든 일이다. 따라서, 코드를 제대로 만드는 것은 단순히 컴파일러의 오류를 제거하는 것으로 끝나지 않으며, 여러 사람의 눈을 통하거나 작은 단위의 테스트를 동반하는 것이 개발에 큰 도움이 된다.

```
void function() {
    int a[10];
    ...

    for (int i = 0; i <= 10; i++) {
        a[i] = 0;
    }
}
```

위의 프로그램은 아주 단순하다. 하지만, 약간의 실수로 인해서 전체 프로그램이 망가질 위험이 있다는 것을 알 수 있을 것이다. 즉, 배열의 크기를 10으로 설정했지만, `for()` 루프에서 0에서 10까지 11개의 배열의 요소에 접근하기 때문이다. 이와 같은 사소한 오류는 그나마 금방 찾아내기 쉽지만, 다음과 같은 경우는 쉽지 않다.

```
if ( a == '1' && b == '2' || c == '3' ) {
    return 0;
}
```

이 코드에서는 "a"라는 변수를 사용한 목적이 '1'을 설정하기 위한 것인지, 아니면 '1'과 비교를 위한 것인지가 모호하다. 컴파일러의 입장에서는 이런 것을 만날 때, 프로그래머의 의도를 알 수 없기 때문에, 최대한 문법적으로 정확하다면 아무런 경고 메시지를 나타내지 않을 가능성이 있다. 프로그램이 잘못되었다는 것을 즉각적으로 알 수 있으면 이러한 오류는 금방 검출 할 수 있지만, 다른 코드들과 섞여있거나, 혹은 이름이 복잡하거나 긴 변수와 같이 사용된다면, 눈으로 찾아내기는 힘들다.

따라서, 위와 같은 경우를 미연에 방지하기 위해서는 두 가지 방법으로 접근하는 것이 필요하다. 첫 번째 방법은 비교문을 사용할 때는 변수의 위치를 변경하는 방법이다. 즉, "a == '1'"과 같이 사용하지 않고, "'1' == a"와 같이 사용한다면, 컴파일러가 "'1' = a"를 잘못된 것으로 생각해서 컴파일 오류를 발생시킬 것이다. 두 번째 방법은 작성 후에 테스트를 만드는 것이다. 즉, "a"값을 "'1'"이나 다른 값으로 설정해서 정확히 동작하는지를 실행해서 확인하는 방법이다(이를 단위 테스트(Unit Test)라고 함). 첫 번째 방법의 문제점은 일반적으로 사용하는 문법이 아니기에 가독성을 떨어뜨릴 가능성이 있다. 두 번째 방법의 문제점은 단위 테스트가 생략하는 경향이 있다는 점이다. 따라서, 둘을 함께 조합하는 것이 좋긴 하지만, 일단은 후자의 방법이 더 확실한 검증 방법이라고 할 수 있다(주로 코딩에서는 첫 번째 방법을 선호).

눈으로 코드를 더 잘 보이게 표현하는 방법은 공백을 사용하는 방법과, 일반적으로 표현되는 것을 그대로 유지하는 방법이 있다. 라인 단위의 공백은 하나의 블록("}")으로 나누어지거나, 혹은 논리적인 내용의 연속이 구분될 필요가 있을 때 넣는 방법이다. 하나의 라인 사이의 공백은 연산자와 연산 대상이 되는 것을 구분하고, 괄호와 그 다음에 오는 단어사이에 공백을 넣는 것이다. 눈에 잘 들어오면 그만큼 오류도 찾기 쉬워진다.

```
int a = 1;
int b = 2;
int c = 3;

for (int i = 0; i < 10; i++) {
    ...
}

printf("%d, %d, %d\n", a, b, c);
```

위의 코드와 같이 변수의 선언부와 일을 처리하는 부분, 출력이 이루어지는 부분은 나누어서 생각해 볼 수 있는 논리적인 묶음이라고 보이며, 연산자와 연산자 사이에 공백은 더 뚜렷하게 무엇을 하려고 하는지를 보여줄 수 있다. `for()` 루프의 경우에도 일반적인 문법의 사용을 가정한 것이다. 따라서, 눈에 잘 들어오는 부분에 대해서 크게 신경쓰지 않고도, 코드를 더 잘 볼 수 있도록 만들어 준다.

`switch()`문의 경우에는 흔히 생기는 문제점은 "break"가 생략된 `case`문과 "default"가 없는 경우이다. 따라서, 습관적으로 "break"와 "default"는 항상 사용하는 것이 좋다. 물론, 그렇게 쓰지 않아야 할 경우도 있는데, 그 때는 "break"를 주석(comment)화하는 방법으로 처리하는 것이 코드를 이해하는데 도움이 된다(실수도 줄여줌).

```
switch( a ) {  
    case 0 :  
        ...  
        break;  
    case 1 :  
        /* break가 없음 */  
    case 2 :  
        ...  
        break;  
    default :  
        ...  
}
```

위와 같이 표현된 switch()문은 개발자의 의도를 다른 사람에게 명확히 전달해 줄 수 있어서, 향후에 발생하는 코딩오류를 미연에 방지할 가능성이 높다. 중요한 것은 코드의 가독성을 높이면 그만큼 코드의 오류가 줄어듦과 동시에, 코드를 더 빨리 변경할 수 있도록 만든다는 점이다. 따라서, 코딩의 목적은 효율적인 코드를 짜는 것도 있지만, 읽기 쉽고 이해하기 쉬운 코드를 만드는게 핵심이라고 할 수 있다.

2. 코딩 룰

코딩 룰은 코딩을 시작하기 전에 마련하는 것이 좋다. 보통의 경우 팀 단위로 달리 가져가기도 하고, 혹은 과제 별로 다르게 할 수도 있다. 중요한 점은 일관성을 가지고 모든 개발자들이 같이 사용하는 규칙을 만드는 것이다. 또한, 너무 많은 규칙보다는 빨리 실무에 적용할 수 있는, 중요한 것들을 위주로 해서 정하는 것이 좋다. 새롭게 C언어에 입문한 사람이라면, 좋은 코딩을 익히는 방법으로 남들이 만들어 놓은 규칙을 따라하는 것도 좋을 것이다.

[코딩 룰(Coding Rule)이란?]

일반적으로 코딩 룰(Coding Rule)은 코딩 표준(Coding Standard), 코딩 스타일(Coding Style), 코딩 컨벤션(Coding Convention) 등으로 표현된다. 이와 같은 코딩 룰은 주로 과제 단위나, 혹은 팀이나 회사 차원에서 정의되어 사용하며, 코딩의 산출물인 소스 코드(Source Code)를 만드는 방식을 정의한다. 원칙은 여러 사람이 작업을 하더라도 한 사람이 짠 것처럼 보이도록 만드는 것이다. 이렇게 함으로써 얻는 효과는 코드의 가독성을 확보할 수 있다는 점이다. 즉, 코드의 직접적인 개발에 참여하는 인력 및 그 외 코드를 읽는 사람들에게 일관성을 제공해서, 코드를 쉽게 이해할 수 있도록 만든다. 개발자들이 각자 자기만의 특징을 가지는 코드를 만든다면, 보는 사람의 입장에서 이해하기가 어려운 코드로 보일 것이다. 코드는 작성하는 사람은 한 명일지라도, 읽는 사람은 여러 사람이 될 수 있다.

01. 변수는 선언과 동시에 초기화를 한다.
02. 모든 함수는 반드시 복귀 값을 돌려주도록 정의한다.
03. 모든 함수는 자신이 원하는 값을 인수로 해서 호출 되는지 검사한다.
04. 모든 함수는 자신의 역할을 설명하는 이름을 가지도록 한다.("동사+명사"를 기본으로 하며, "_"와 같은 것을 가질 수 있다. 첫 번째 문자를 소문자로 하며, "_"이 없을 경우에는 두 번째 단어 부터는 대문자를 시작 문자로 사용한다. "_"을 사용할 경우에는 소문자 만을 쓰도록 한다.
05. 모든 함수 내의 임시 변수들에도 의미있는 반드시 이름을 주도록 한다. 예외적으로, 루프를 위한 "i", "j", "k"와 같이 일반적으로 인덱스(Index)로 많이 사용되는 경우에는 그대로 사용할 수 있다.
06. 함수는 짧아야 하며 한 가지 일만 수행해야 한다. 함수 내부의 코드는 함수의 이름보다 한 단계 아래의 추상화 수준으로 구현 되어야 한다.
07. "매직 넘버(Magic Number)"가 없어야 한다. 필요한 경우에 "enum"이나, "define" 혹은 "const int"와 같은 것을 사용해서 이름을 반드시 주어야 한다.
08. 함수는 인수(parameter)가 없는 것을 기본으로 한다. 가능한 최소한의 인수만을 사용해야 한다(5개 이상인 함수는 만들지 않는다).
09. 주석은 이유를 설명하기 위해서만 사용한다. 하는 일을 표현하기 위해서는 코드를 사용 한다. 코드에서 직접적인 이유를 설명하지 못하는 경우에만 주석을 사용한다.
10. 코드 간에는 상호 호출이나 상호 참조를 만들지 않는다. 반드시 단방향 호출이나 참조만 존재하도록 만든다(콜백(Callback)함수는 단방향 참조다. 호출하는 측에서는 설정된 것만 이용할 뿐이다.)

일단은 이와같이 10가지만을 정의하도록 하자. 나중에 더 추가할 것이 있으면 그 때 고치면 된다. 이와 같은 코딩 룰은 간단하면 간단할 수록 좋다. 하지만, 모든 것이 그렇게 쉽지는 않다. 잘 조직화된 팀은 (Self-Organized Team)은 이미 많은 부분에 있어서 암묵적인 규칙을 가지고 있을 것이다. 그런 팀은 많은 코딩 룰이 필요치 않다. 하지만, 새로 과제를 시작할 때 새롭게 구성된 팀은 좀 더 많은 규칙이 필요할 것이다. 그리고, 정말 중요한 것은 룰이 아니라 그것을 일관되게 과제의 완료 시점까지 지키는 것이다. 정의할 때는 이견이 있을 수 있지만, 한번 정리된 후에는 그것을 꾸준히 따라야 한다. 필요하다면 룰에 따르는지도 (룰을 사용해서) 검증할 수 있어야 할 것이다.

[명료한 코드 만들기]

코드를 읽기 쉽게 만드는 것이 코딩의 핵심이다. 여기서 "읽기 쉬운"이라는 말의 뜻이 중요하다. 흔히 소프트웨어 개발자들이 생각하는 읽기 쉽다는 말의 뜻은 다음과 같이 요약해 볼 수 있다.

- 짧아야 한다.
- 변수의 이름이 의미를 가져야 한다.
- 여러 다른 코드를 접근하지 않아야 한다.
- 조건문의 분기가 적어야 한다.
- 잘못 해석할 가능성이 없어야 한다.

이외에도 몇 가지 더 생각나는 것들이 있을 수 있겠지만, 아마도 위에서 나열한 것들이 가능 기본일 것이다. 그렇다면, 자신이 만드는 코드와 위에서 말한 것들을 비교해 보고 결과를 정리할 수 있을 것이다. 아마도 대부분의 개발자는 생각보다 읽기 쉬운 코드를 만드는 것이 쉽지 않다는 것을 발견할 것이다.

물론, 어떤 개발자들은 자신의 코드를 보면서 느끼는 감정이 "정말 잘 만들었어"와 같은 자화자찬일 수도 있다. 자신이 그렇게 생각한다고 해서 남들도 그렇게 생각할까? 대부분의 경우 자신에게는 후한 점수를 주지만, 타인에게는 엄격한 것이 사람의 본성이다. 그리고, 자신의 코드는 잘 이해가 된다고 생각하지만, 남이 만든 코드에 대해서는 불신하는 것이 일반적이다. 따라서, 우리가 할 수 있는 최선의 길은 "나의 시각을 버리고, 다른 사람의 비평을 듣는 것"이 될 수 밖에 없다. 다른 사람이 이해할 수 있을 정도라면 충분히 쉽게 작성한 코드라고 생각해도 된다. 물론, 그렇다고 위에서 말한 5가지 기본적인 것들이 필요 없다는 것은 아니다. 즉, 코딩할 때는 기본적인 부분들을 잘 지켜야만 전체적인 코드의 완성도가 높아질 수 있다.

긴 코드

- 이해하기 힘들다.

1. 사용되는 변수가 많다.
2. 복제된 코드가 발생한다.
3. 조건문이 길어진다.
4. 자료구조가 복잡하다.
5. 의존성이 높아진다.
6. 여러가지 역할을 한다.
7. 파라미터의 개수가 많다.
8. 최적화가 어렵다.
9. 수정이 자주 발생한다.
10. 버그가 자주 발생한다.
11. 테스트가 어렵다.
12. 구조화 시키기 어렵다.
13. 실행 안되는 코드가 있다.
14. 주석 처리된 코드가 있다.



짧은 코드

- 이해하기 쉽다.

1. 사용되는 변수가 적다.
2. 코드의 재활용이 쉽다.
3. 조건문이 짧아진다.
4. 자료구조가 단순하다.
5. 의존성이 낮아진다.
6. 한가지 역할만 한다.
7. 파라미터의 개수가 적다.
8. 최적화가 쉽다.
9. 수정이 가끔 발생한다.
10. 버그가 가끔 발생한다.
11. 테스트가 쉽다.
12. 구조화 시키기 쉽다.
13. 실행 안되는 코드가 없다.
14. 주석처리된 코드가 없다.

"짧은 코드"가 의미하는 것은 언어가 가지는 문법을 아주 극단적으로 사용 하라는 의미가 아니다. 즉, 논리적인 추상화 수준에서 한 가지 일만하는 함수를 짧게 만들라는 뜻이다. 예를 들어, 100라인의 함수 보다는 15라인의 함수가 더 이해하기 쉽다. 짧으면 짧을수록 이해하고 수정하는 것이 쉬워지며, 수정해야 하는 폭도 줄어들게 된다. 물론, 그것이 오버헤드라고 생각할지도 모르지만, 작은 희생으로 큰 이익을 볼 수 있다는 생각은 지울수 없다. 그리고, 그 희생 자체도 지극히 작을 수 있다.

변수를 선언할 때 수학의 방정식과 같이 "x, y, z"와 같이 이름을 준다면, 변수의 역할에 대해서는 아무것도 알려주지 못하게 된다. 즉, 등장 인물의 이름으로 그 역할에 대해서 짐작할 수 있는 정보는 사라져 버린다. 예를 들어, "말자(개똥이)"와 같은 이름을 가진 여자라면, "마지막에 태어난 여자 아이(귀여운 막내동이)"와 같은 의미가 있겠지만, "xx3"과 같은 이름은 xx염색체를 가진 3번째로 해석할 수도 있고, 뭔가 나쁜 단어를 감추기 위해서 사용한 것 처럼도 보일 수도 있다. 따라서, 변수의 이름을 줄 때는 그 변수의 역할을 유추할 수 있는 정보도 함께 포함하는 것이 좋다.

만약, 하나의 함수가 여러 다른 함수들을 호출하고 있는 관계를 가진다면, 호출이 많을수록 테스트하기는 힘들다. 즉, 단위 테스트의 대상이 되는 함수가 다른 함수들에 의존 하기에, 테스트 하기 위해서는 전체 조건을 설정해야 하는 부분이 많아지게 된다. 물론, 이런 것이 전혀 발생하지 않을수는 없다. 따라서, 가능한 줄이는 것이 좋다는 말이다. 특히, 최상위에 위치한 함수가 절차적으로 다양한 함수를 호출해서 일을 처리해야 하는 경우가 여기에 해당한다. 이런 함수들은 하위 함수들이 다 구현된 이후에 사용될 수 있다. 여기서 중요한 것은 가능한 자신이 처리해야 하는 일과 남이 처리해 주어야 하는 일을 구분하고, 역할을 벗어나는 일에 대해서는 요청(Request)를 보내는 방식으로 처리해야 한다. 정리하면, 가능한 연결의 고리를 줄이는 것이 문제 발생 가능성을 줄여 준다.

조건문의 분기는 생각의 흐름을 멈추게 만든다. 따라서, 조건문이 여러 개가 있다면 각각에 대해서 생각을 멈추고 여러가지 상황을 고려하는 시간이 필요하다. 따라서, 잘 읽혀지는 코드를 만들기 위해서는 조건문을 최소화 하는 것이 좋다. "if()"만이 조건문은 아니며, "switch()"와 같은 경우는 더 나쁜 상황을 만들 수 있다. 중첩된 조건문은 여기에 한 번 더 생각하도록 요구하기에 상황을 더 악화시킨다. 가능한 간단한 조건문을 만들고, 조건 자체는 "True"나 "False"를 가질 수 있는 "조건문"만 있어야 한다(할당은 없어야 한다). 그렇지 않다면, 언어의 문법이 지원하는 것 이외에 "임의적인 변환"이 사용되거나 내포된 의미로 접근할 것이기에, 생각 역시 더 복잡해 지게 된다.

가장 중용한 것은 코드를 실수로라도 잘못 해석할 가능성을 원천적으로 없애는 것이다. 명료하게 작성하라는 뜻이다. 예를 들어, "if()"절에 "{}"와 같은 것이 없는 단문인 경우도 문제가 될 가능성이 있다. 추가적인 코드의 삽입이나 수정이 발생할 경우, 잘못될 가능성은 충분히 있는 것이다. 따라서, "{}"를 사용해서 명확히 블록화 시킨 코드로 만들어 주는 것이 좋다. 물론, 컴파일러는 이런 경우에도 원래의 의미를 오해석하는 경우는 없다. 사람이 실수할 수 있는 것을 기계적으로 막아주는 것도 좋은 방법이다. 예를 들어, "="와 "=="를 실수하는 경우도 있기에, 이때는 상수와 변수의 순서를 바꾸는 것도 고려해 볼 수 있다. 중요한 것은 코드를 읽는 사람이 실수할 수 있는 가능성을 최소화 하는 것이다. 실수는 누구나 한다. 하지만, 반복된 실수는 실력일 뿐이다.

[코딩 룰(Coding Rule) 자세히 보기]

코딩 룰(Rule)은 두 가지 측면에서 중요하다. 첫 번째는 코드를 더 읽기 쉽게 만드는 목적이 있으며, 두 번째는 버그를 예방하는 효과를 가진다. 코드를 더 읽기 쉽게 만들기 위해서는 일관성을 가지고 작성해야 한다는 말이며, 그 일관성의 목표가 읽기 쉬운 코드를 만드는데 있다는 것이다. 즉, "읽기 쉬운 코드가 좋은 코드"라는 일반적인 생각에 근거한다.

모든 프로그래밍 언어가 마찬가지겠지만, 사용자의 의도를 완벽히 이해하는 컴파일러는 없다. 따라서, 사용자는 자신의 의도를 프로그래밍 언어로 표현하는 일에 종종 실패할 수 밖에 없다. C언어는 문법적인 오류가 없다고 판단되면 대부분의 것들을 사용자의 책임으로 남겨둔다. 이 말은 사용자가 제대로 사용하

지 못하면 쉽게 프로그램을 망칠 수 있다는 뜻이다. 따라서, 이런 경우를 미연에 방지할 수 있는 코딩에 관련된 규칙을 아는 것은 중요하다.

유명한 코딩 룰로는 MISRA-C와 같은 것이 있으며, 현재는 2012버전까지 나온 상태다. 주로 자동차와 같은 인명의 손실과 같이 위험이 큰 프로그램을 코딩할 때는, 표준에서 제공하는 방법으로 코딩하는 것을 적극 검토해야 하는 것이 좋다. 여기서는 MISRA-C와 같은 코딩 룰을 보는 것이 아니라, 일반적으로 알려진 좋은 C언어의 코딩 습관들을 나열해 보도록 하겠다. 특히, C언어를 사용해서 임베디드 시스템을 만들 때 적용해 볼 수 있는 것들이다.

가정 : 모든 코드는 개인 소유가 아닌 협업의 산물이며, 개발자들은 코드를 깨끗한 상태로 유지해야 할 책임이 있다.

Rule 00. 컴파일러의 경고 수준을 가장 높여서 컴파일해야 하고, 경고 메시지가 없어야 한다.

; 이건 소프트웨어를 개발하는 개발자라면 상식 수준의 일이다. 하지만, 실제의 코드에서 이런 문제는 빈번하게 발생한다. 만약, 다른 개발자가 코드를 받아서, 컴파일 옵션을 수정한 후에 컴파일을 한다면, 화면에 나오는 컴파일러의 경고 메시지를 보는 순간, 코드에 대한 신뢰도는 급속히 떨어질 것이다. 코드를 믿을 수 없다는 것이다. 따라서, 당연히 코드에서 발생하는 경고 메시지는 배포 전에 반드시 제거되어야 한다. 제거가 불가능하다는 불평을 하는 사람들도 있고, 문제가 아니라는 변명을 하는 사람들도 있지만, 그런 문제는 전부 제거가 가능하다. 게으름이 문제지 시간 낭비는 결코 아니다. 이건 다른 코딩 룰들이 의미를 가지기 위해서 기본적으로 지켜야 할 개발자의 의무다. 경고 옵션을 낮추어서 메시지를 잠시 보이지 않게하는 것은 해답이 아니다.

gcc -Wall

Rule 01. 하나의 라인에는 한 문장(Statement)만 남겨라.

; 한 덩어리로 이해될 수 있는 경우를 제외하면, 하나의 라인에는 하나의 문장을 남기는 것이 원칙이다. 여러 문장을 남겨두면 이해하기도 어렵고 실수할 가능성도 높다. 변수의 선언과 같은 경우에도 한 라인에는 하나의 변수만 선언해 주는 것이 좋다.

```
int a = 10;
unsigned short b = 15;
if (x == 10) return xxx; /* 충분히 이해하기 쉽기 때문에 이렇게 처리했다. */
```

Rule 02. 블록화할 수 있는 코드들은 "{ }"으로 묶어서 사용하라.

; "if()", "while()", "switch()", "for()" 등등 블록으로 실행되어야 할 코드들은 전부 "{}"를 사용하는 것이 좋다. 한 문장만 있는 경우라도 "{}"를 빼먹지 사용하는 것이 나중에 버그를 찾아내거나, 추가로 코드를 작성할 때 오류를 예방할 수 있다.

```
if ( x == 10 ) {
    do_something();
}
```

Rule 03. 연산자의 우선순위에 의존하지 말고 "()"를 사용하라.

; 어떻게 계산될지 완벽히 알 수 없다면, 연산자의 우선순위 보다는 "()"를 사용하는 것이 오류를 만들지 않는다. 특히, 구조체에 대한 포인터를 사용할 경우, 필드를 접근하기 위해서는 포인터 변수가 가르키는 내용을 표현하기 위해서 반드시 "()"를 사용해야 한다(대신에 "->"을 사용하는 것을 권장한다.).

```
(*pointer).field = 100;
pointer->field = 100; /* 포인터를 이용한 구조체 필드는 이렇게 사용하는 것이 좋다. */
```

Rule 04. “static”, “const”와 같은 것을 반드시 사용하라.

; 함수를 정의할 때 “static”과 “const”를 명시적으로 사용하는 것은 함수의 책임 범위를 한정해 주며, 함수를 사용하는 측에 더 많은 정보를 준다. 내부 정보를 공개하는 것이 아니라, 계약으로 정한 인터페이스에 대한 정확한 사용 방법을 알려준다는 측면에서 적극적으로 사용할 필요가 있다. “static”으로 정의된 함수의 경우 함수를 사용할 수 있는 범위가 파일 내로 한정되며, 외부에 노출되는 것을 막아준다. 이는 파일의 내부에 정의된 함수만 접근할 수 있다는 것을 명시적으로 표시해 준다. 함수의 파라미터를 “const”로 정의하는 것은, 함수가 전달 받은 파라미터에 대해서 변경하지 않는다는 것을 알려준다. 따라서, 사용하는 측에서는 데이터의 변경이 발생하지 않을 것이라는 믿음을 가질 수 있다.

```
static void function(const int x, const unsigned int y) {
    ...
    return;
}
```

Rule 05. 전역변수(Global Variable)를 사용하지 말라.

; 전역변수를 사용하는 이유는 성능을 개선하기 위한 것이라고 이야기 하지만, 대부분의 경우 성능보다는 개발 편의를 위한 경우다. 또한, 설계 미흡으로 역할과 책임의 분리가 제대로 되지 않은 모듈을 만들었기 때문이다. 전역변수는 컴파일리의 최적화에 방해(레지스터를 변수가 사용하지 못하고, 매번 메모리에서 읽어야 함)가 되며, 모듈들이 서로 의존하도록 만든다. 더 안좋은 것은 그렇게 의존적인 부분이 어떻게 변경될지 알 수 없으며, 그것이 다른 모듈에 어떤 영향을 줄지도 모른다는 점이다. 따라서, 전역변수를 안쓰는 것이 원칙이다. 만약, 전역 변수와 같은 것이 필요하다면, 데이터를 다루는 모듈을 따로 정의해서 인터페이스를 통해서 변수에 접근하는 방식으로 코딩해야 한다.

```
typedef struct my_struct {
    unsigned int value;
} MY_STRUCT;

MY_STRUCT my;

unsigned int getValue() {
    return my.value;
}

void setValue(unsigned int value) {
    my.value = value;
    return;
}
```

Rule 06. 더 작은 범위를 가지는 값으로 타입 변환(Type Casting) 하지 말라.

; 타입 변환은 근본적으로 위험을 가지고 있다. 특히, 더 작은 자료형으로 타입 변환을 하는 것은 유효한 값을 잃어버린 가능성이 높다. 따라서, 가능한 타입 변환을 하는 것 보다는 원래의 형으로 사용하는 것을 권장한다. 메모리 할당과 같은 경우는 어쩔 수 없이 타입 변환을 해야 하는 경우이며, "void *"으로 선언된 구조체의 마지막 필드에 대한 것도 타입 변환이 허용된다. 이와 같이 타입 변환을 허락할 수 있는 부분을 정해서 사용하는 것이 도움이 되지만, 근본적으로 타입 변환을 될 수 있으면 하지 않는 것이 좋다. 즉, 정의한 타입으로 변수를 사용하는 것이 가장 좋다.

```
char *temp;
if ( temp = (char *)malloc(sizeof(MY_STRUCT))) == NULL) {
    do_something();
```

```

}
```

Rule 07. “register”, “auto”, “goto” 등의 키워드 사용하지 말라.

; 대부분의 경우 사람보다 컴파일러가 최적화를 더 잘 한다. 물론, 컴파일러도 사람이 만드는 것이지만, 컴퓨터에 대한 이해가 깊고 전용으로 만들어진 컴파일러는, 사람이 의도한 최적화보다 더 나은 최적화를 할 가능성이 높다. 물론, “volatile”과 같은 키워드로 특정 변수가 부지불식간에 변경이 될 가능성이 있어서 최적화를 막아야 할 필요가 있다는 것을 알려주어야 할 때도 있다. 하지만, 컴파일러에서 제공하는 최적화 옵션과 코드 생성에 대해서 개입하지 않는 것이 좋다. 최적화를 위해서 코딩하지 말고, 구조화 시키기 위해서 코딩하는 것이 최선이다.

```

void function(const int x, const int y) {
    int a; /* register int a */
    ...
}
```

“goto” 문에 대해서는 엇갈린 시각이 있지만, 기본적으로 사용하지 않는 것이 규칙이다. 갑작스럽게 제어가 옮겨지는 것은, 개발자의 의도를 이해하지 않으면 코드를 수정하기 어렵게 만들 수 있다. 또한, 자원에 대한 할당이 있을 경우에는, 제대로 해제를 하지 않아 자원 누수(Resource Leak)이 발생할 수도 있다. “goto”문은 항상 동일한 다른 방법으로 처리할 수 있기에, 대체 수단을 이용하는 것이 좋다. 참고로, “continue”와 “break”的 경우도 조심해서 사용해야 하지만, 여기에는 포함하지 않았다.

Rule 08. 포인터를 나타내는 “*”는 타입에 붙여서 사용하라.

; 타입 선언에 붙여서 “*”를 선언하는 것이 포인터라는 것을 명확히 더 잘 보여줄 수 있다. "char *ch"보다는 "char* ch"을 사용해야 할 것이다. 눈에 잘 보인다는 것은 그 만큼 실수를 줄일 수 있다는 것이다.

```

#include <stdio.h>
#include <stdlib.h>

int main(void) {
    char* ch, myCh; /* myCh가 잘못 선언되었다. */
    /* 여기에 “char* myCh;”를 추가하는 것이 더 좋다. */

    puts("Pointer Declaration Example 01");
    ch = "Hello, World!!!\n";
    myCh = "Hello, World!!!\n"; /* 컴파일에서 경고 메시지를 출력할 것이다. */

    printf("%s", ch);
    printf("%s", myCh); /* 컴파일에서 경고 메시지를 출력할 것이다. */

    return EXIT_SUCCESS;
}
```

위의 코드는 두 가지 문제가 있다. 첫 번째는 하나의 라인에 두 개의 변수가 선언되었다는 것이다. 두 번째는 각각이 같은 타입이 아니라는 점이다. 만약 “myCh”앞에 있는 “*”가 없다면, 의도한 출력을 만들지 못할 것이다.

Rule 09. 주석은 “/* */”만 사용하라.

컴파일러에 따라 “//”와 같은 한 줄 짜리 주석을 허용하는 경우도 있으나, 이런 주석들은 대부분 의도적인 것이 아니라 급하게 만들어지는 경향이 있다. 주석보다는 코드를 제대로 읽히게 만드는 것이 더 중요한 일이며, 한 줄 짜리 주석보다 한 줄의 이해하기 쉬운 코드가 더 바람직하다. 개발자가 편의로 “//”을 사용

해서 코드를 주석으로 만드는 경우도 있으며, 그런 코드들은 전부 배포 전에 제거되어야 한다. 주석으로 된 코드를 발견하면 주저할 필요없이 삭제를 권장한다. 참고 용도로 남겨 놓았다고 이야기 한다면, 그냥 버전 관리 시스템을 쓰라고 하라.

```
// temp = xxx + yyy * zzz; ---> 삭제할 것.  
// 2016.05.08 SH Kwon Edited —> 삭제할 것.
```

참고로 주석을 이용해서 코드를 문서화하는 경우라면, 주석이 코드와 일치하는지 확인해야 한다. 코드의 변경이 주석과 다를 경우, 코드를 읽는 사람에게 난해한 코드가 되기 때문이다. 따라서, 주석이 붙어 있는 코드의 경우에는 코드 리뷰에서 코드만 봐서는 안되며, 주석도 같이 리뷰가 할 수 있도록 해야 한다.

Rule 10. 탭(Tab)보다는 빈칸(Space)을 사용해서 들여쓰기(Indentation) 하라.

탭으로 만들어진 들여쓰기는 다른 편집기나 운영체제(윈도우즈 혹은 유닉스)에서 다르게 보일 가능성이 있다. 따라서, 탭보다는 빈칸을 사용하는 것이 코드를 일관되게 보여주는데 도움이 된다. 물론, 이것이 버그와 직접적인 관련을 가진 것은 아니지만, 코드를 좀 더 명확하게 보일 수 있도록 만들어 코드의 이해를 높여줄 수 있다. 당연히 잘 읽히는 코드는 버그가 줄어들 것이다. 최근의 편집기들은 들여쓰기를 탭이나 빈칸 중에서 선택해서 사용할 수도 있고, 빈칸의 갯수도 조절할 수 있다. 들여쓰기는 4 혹은 8정도면 적당하게 사용할 수 있다. 한 문장의 길이는 최대 140 칼럼 이상은 가지 않는 것이 좋다. 당연히 더 짧을수록 더 이해하기도 쉽다.

```
void function( void ) {  
    xxx;  
}
```

Eclipse 와 같은 편집기는 코딩 스타일을 일괄적으로 적용할 수 있기에, 따로 빈 칸에 대한 것을 신경쓰지 않아도 된다. 잘 사용되는 스타일로는 "K&R", "BSD" 등이 있으니, 선택해서 사용하면 된다. 필요한 경우 코딩 스타일의 일부를 팀원들의 동의 하에 적절히 수정해서 사용할 수도 있다.

Rule 11. 조건부 컴파일을 사용해서 모델의 분기나 기능 추가를 하지 말라.

; 모델의 분기나 조건을 추가하는데 조건부 컴파일("#if ... #else .. #endif"등)을 사용하면, 점차 이해하기 어려운 코드로 변한다. 빌드(Build) 스크립트도 고치기 힘들며, 어떤 조건을 컴파일 시에 명시적으로 전달해야 할지도 혼동하게 된다. 따라서, 조건부 컴파일은 될 수 있으면 사용하지 않는 방향으로 코딩해야 한다. 필요하다면, 파일을 분리하거나 계층을 정의해서 모델이나 기능의 분기에 안정적으로 대처할 수 있는 방법을 생각해야 할 것이다.

```
void function( int k ) {  
  
#ifdef __MODEL_X__  
    int x = k;  
#endif  
#ifdef __MODEL_Y__  
    int y = k;  
#endif  
...  
    return;  
}
```

개발 초기에는 모델의 분기를 시키지 않더라도, 기능의 추가나 후속 모델의 개발 등에서 조건부 컴파일을 사용해서 쉽게 확장 하려는 유혹을 느끼게 된다. 이런 식으로 사용 되는 코드가 여러 곳 정의되어 있

다면, 이를 모아서 새로운 계층과 인터페이스로 만들 수 있다. 즉, 특정 모델이나 기능에 의존하는 코드를 인터페이스에 의존하는 방식으로 분리해서 변경에 대해 대처할 수 있게 된다.

디버깅 목적으로 조건부 컴파일을 사용하는 것은 타당하다. 즉, 디버그 메시지를 출력하기 위해서 `#ifdef __DEBUG__ #endif`와 같이 코드를 사용할 수 있다. 나중에 배포를 위한 목적인 경우에는 이렇게 조건부로 컴파일 가능하게 만든 코드들이 전부 빠지는지 확인해야 할 필요가 있다. 가끔 디버깅 목적으로 만들어진 코드들이 배포되는 코드에 섞이게 되는 경우도 있기 때문이다.

Rule 12. 각각의 블록("{}")의 앞과 뒤에는 빈 라인을 두도록 한다.

; 논리적으로 분리된 것으로 생각되는 블록의 앞 뒤에는 빈 라인을 두는 것이 코드를 보기 편하게 만든다. "if()"나 "for()"등과 같이 블록을 가진 조건문 등에도 같은 규칙을 적용할 수 있다.

```
if ( x == k ) {  
    ...  
}  
  
temp = x;  
...
```

만약, 논리적으로 이어지는 블록에 속한 문장인 경우에는 블록과 함께 읽혀지기를 원할 것이다. 따라서, 블록의 시작 보다는 논리적으로 같이 묶여있는 부분의 위에 빈 라인을 두어야 할 것이다.

```
int i = 0;  
for ( ; i < MAX; i ++ ) {  
    ...  
}
```

Rule 13. 모든 변수의 이름은 소문자를 사용한다.

; 소문자는 주로 변경될 수 있는 곳에 사용하고, 대문자는 변경이 되지 않는 곳에 사용하는 것이 코드의 이해를 편하게 한다. 상수 값과 같은 것은 대문자로 표기해서 코드에서 즉시 알아 볼 수 있도록 만드는 것이 좋다.

```
#define PI 3.141592  
enum {  
    MODEL_A = 0,  
    MODEL_B,  
    ...  
};
```

Rule 14. 사용자 정의 타입에 대해서는 대문자를 사용하도록 한다.

; 사용자가 선언한 타입은 시스템에서 기본적으로 제공하는 타입이 아니다. 따라서, 명확하게 보여줄 필요가 있다. 구조체와 같은 것은 "typedef"를 사용하지 않을 경우에는 소문자로 선언해도 되지만, "typedef"로 일반적으로 제공하는 타입과 같은 형태로 사용할 경우에는 대문자로 표현해 주어야 할 것이다.

```
typedef struct mystruct MYSTRUCT;  
...  
MYSTRUCT* my_type;
```

Rule 15. 함수의 이름은 자신이 사용하고 있는 시스템의 룰을 따르도록 한다.

; Windows와 Linux는 서로 다른 형식으로 함수의 이름을 표기한다. Windows의 경우에는 "HungaryNotation"과 같은 방법을 사용하며, Linux의 경우에는 모든 함수의 이름을 소문자와 "_(Underscore)"로 만든다. 즉, "void my_linux_function()"와 같은 형식이다. 따라서, 자신이 사용하는 운영체제 환경에서 기본적으로 제공하는 이름을 만드는 방법을 사용하는 것이 코드를 더 일관되게 표현하는 방법이다.

만약, 특별한 운영체제를 가지고 있지 않거나, 혹은 다양한 운영체제를 지원할 목적을 가지고 있는 코드를 만든다면, 둘 중에 한 가지를 선택해서(후자를 선호하는 편이지만), 일관되게 사용하면 된다. 함수나 변수의 이름에 약자(Abbreviation)를 사용하지 않는 것이 좋지만, 일반적으로 업계에 알려진 약자는 사용해도 된다. 약자를 사용하면 처음 접하는 사람에게 고민거리가 되기 때문이다.

함수의 이름과 변수의 이름 등에 대한 길이 제한은 없으나, 충분히 의미를 전달할 정도의 이름을 가지고 있어야 한다(코딩 룰은 31자를 한정하는 경우도 있다). 컴파일러마다 의미있는 이름의 크기가 한정된 경우가 있기 때문이다). 사용하는 목적을 묘사적으로 표현하는 이름이면 좋다. 함수의 이름은 "동사_명사()"와 같은 형태가 좋으며, 변수는 "형용사_명사"와 같은 형태가 좋다.

Rule 16. 함수의 길이는 100(더 짧아도 되지만)라인을 넘어선 안된다.

; 짧은 함수가 이해하기 쉽고 고치기 쉽다. 따라서, 긴 함수를 만드는 대신 짧은 함수를 많이 만드는 것이 더 이해하기 쉬운 코드를 만든다. 함수가 짧아지면 재활용하기도 쉽게 되며, 파라미터의 개수도 적어진다. 당연히 함수의 내부에서 사용되는 변수들도 레지스터(Register)를 할당받을 가능성이 높다. 복잡한 일을 복잡하게 해결하는게 코딩을 잘 하는 것이 아니라, 복잡한 것을 간단하게 만들어서 해결하는 것이 정말 코딩을 잘하는 것이다.

함수는 오버헤드를 가진다. 즉, 호출에 관련된 오버헤드다. 함수는 호출되기 위해서 스택을 설정해야 하며, 사용하고 있던 레지스터에 저장된 값을 스택에 보관해야 한다. 또한, 호출 후에는 복귀값을 레지스터에 저장하고 다시 읽어 들이는 과정이 필요하다. 될 수 있으면 짧은 함수를 만들고, 속도가 문제가 될 수 있다면 인라인(Inline) 함수로 만드는 것을 고려해야 할 것이다. 인라인 함수의 대상이 되는 코드는 자주 사용되지 않는 코드가 좋다. 짧은 호출이 발생하면 함수를 인라인으로 만들어서 얻는 혜택이 적어지게 되며, 컴파일러가 인라인화 시키는 것을 거부할 수도 있다.

```
void function( void ){
    /* 긴 함수 */
}

-->

static inline function_A();
static inline function_B();
static inline function_C();

void function( void ) {
    /* 짧은 여러개의 함수로 변경 */
    if( function_A() == 0 ) {
        return;
    }
    function_B();
    function_C();
    ...
}
```

규칙은 100라인으로 했지만, 함수의 길이는 짧을수록 이해하기 쉽다. 따라서, 50라인 정도를 최대 함수의 길이로 정해도 된다. 물론, 함수의 길이를 지나치게 짧게하면, 여러 사람이 불평을 할지도 모른다. 일단 권장하는 길이는 대략 100라인 정도지만, 그 이하로 규칙을 정해도 된다.

Rule 17. 한 줄 이상으로 연결되는 매크로(Macro)는 사용하지 않는다.

; C언어에서 매크로는 많은 부수적인 효과(Side Effect)를 가지는 오류를 만들 수 있으며, 디버깅을 어렵게 만들 수 있다. 따라서, 될 수 있으면 매크로 보다는 인라인 함수(Inline Function)를 사용하는 것을 원칙으로 해야한다. 인라인 함수는 일반 함수처럼 사용할 수 있으며, 함수가 가지는 오버헤드도 가지지 않는다. 기본적으로 함수는 추상화를 위해서 권장하는 부분이며, 코드의 재활용 측면에서도 유리하다.

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

```
#include <stdbooh.h>
static inline int max( int a, int b ) {
    return ( a > b ) ? a : b;
}

static inline bool isBigger( int a, int b ) {
    return ( a > b );
}
```

물론, 한 줄 이하라고 해서, 붙여서 길게 사용하면 안된다. 복잡한 매크로는 그 만큼 복잡한 오류를 만들 가능성이 높기 때문이다. 될 수 있으면 매크로의 사용을 줄이는 것이 중요하다. 오히려 자주 사용될 것으로 생각되면, 일반적인 함수로 만드는 것을 고려해 보기 바란다.

Rule 18. 모든 헤더 파일에는 타입만 선언하고, 실제 구현은 구현 파일(.c 파일)에 정의한다.

; 헤더 파일에는 다른 파일들이 필요로 하는 정보를 보여줄 목적으로만 사용하며, 최소한의 정보만 공개하는 것을 원칙으로 한다. 따라서, 모든 실제 구현은 구현 파일로 숨겨야 한다. 즉, 실제 변수의 정의는 구현 파일에 담도록 해야한다. 전역 변수의 사용을 억제하기 위해서는 헤더 파일에 변수를 선언하는 것도 막아야 한다. 하지만, 특정한 경우에는 전역변수를 사용해야하는 경우도 있기에, 일단은 선언 정도를 남기는 것까지는 가능하다. 물론, 전역변수는 가능한 사용하지 말아야 한다.

```
/* In .h file */
extern int x;

/* In .c file */
...
int x;
```

직접 변수에 접근하는 것보다, 함수와 같은 인터페이스를 제공해서 변수에 접근하는 것이 좋다. 하지만, 이때도 정말 그 변수를 직접 접근할 필요가 있는지 먼저 물어야 한다. 만약, 변수를 사용하는 측에서 그 값을 이용해서 직접 연산을 해야 할 일이 있다면, 차라리 함수를 제공해서 연산의 결과만을 얻어가는 방식으로 코딩하는 것이 좋다. 변수나 자료구조는 언제든 바뀔 수 있다는 점에서, 의존성을 가지는 것은 좋은 코딩 방법이 아니다.

```
/* In .h file */
extern int do_something( void );

...
/* in .c file */
#define MAX 65535
```

```
int x;
int do_something( void ) {
    if( x < MAX ) {
        return MAX - x;
    }
    return x;
}
```

Rule 19. 새로운 타입의 선언은 반드시 "typedef"로 시작해야 한다.

; 사용자가 정의해서 사용하는 타입은 "typedef"로 시작하도록 할 수도 있고, 따로 정의해서 변수앞에 사용할 수도 있다. 혼동을 피하고 일관성 및 타입 펑 오류를 방지할 목적으로라면, "typedef"로 정의해서 새로운 타입을 만들어주는 것이 좋다.

```
typedef struct personal_innformation {
    char *name;
    unsigned int age;
    char *address;
    ...
} PERSONAL_INFORMATION, *PERSONAL_INFORMATION_POINTER;

...
PERSONAL_INFORMATION myinfo;
```

사용자가 정의한 타입이라는 것을 알려주기 위해서, 대문자를 사용하는 것이 코드의 이해를 돋는다. 물론, "personal_information_t"와 같이 소문자와 "_t"를 사용하는 경우도 있다. 따라서, 필요하다면 그렇게 정의해서 사용해도 상관없다. 한번 그렇게 사용했다면, 코드 전반에 걸쳐서 일관되게 사용하는 것이 더 중요하다.

Rule 20. 고정된 크기의 정수 타입을 사용하고자 한다면, "stdint.h"를 사용하라.

; 예전에는 "UNIT8 x"와 같은 형태로 정수의 타입과 크기를 정의해서 코드 전반에 걸쳐서 사용하도록 헤더 파일로 만들어 두었다. 이것은 CPU 아키텍처에서 지원하는 정수의 크기가 다르기 때문에 발생했던 문제로, "C99" 표준에서는 "stdint.h"를 도입해서 해결하고 있다. 따라서, 기존의 방식을 사용해도 문제는 없지만, 표준에서 이미 제공하고 있는 것을 사용하는 것이, 코드의 이식성(Portability)을 높여줄 것이다. 물론, "C99" 표준을 지원하는 컴파일러를 사용한다는 가정하에 그렇다는 말이다.

```
#define UNIT8 unsigned short
...
UINT8 x;

#include <stdint.h>
...
uint8_t x;
```

Rule 21. 부호있는 정수(Signed Integer)는 부호없는 정수(Unsigned Integer)와 비교 연산이나 비트 조작연산에 사용하지 말라.

; 부호가 들어간 정수는 부호 없는 정수와 계산에서 문제를 일으키는 경우가 많다. 될 수 있으면 같은 부호를 가지고 있는 값들을 사용하는 것이 오류를 일으킬 가능성을 줄인다. 특히, 비트 조작과 같은 경우에는 예상하지 못한 결과를 보일 수 있으므로, 섞어서 사용하는 것을 주의해야 할 것이다. 변수는 정의된 타입 끼리 연산을 할 때 가장 안정적이다. 형 변환과 같은 것도 될 수 있으면 사용을 자제해야 한다. 물론, 메모리 할당이나, 구조체의 마지막 필드에 할당될 수 있는 배열과 같은 것은 포인터를 이용한 형 변환을

사용해서 접근하는 것이 허락된다.

```
unsigned int x = 100;
signed int y = -200;

if( x | y ) {
    printf("Hello? Don't use bit field operation with signed integer!!!");
}
```

Rule 22. 함수의 복귀 시점 가능한 빨리 할 수 있도록 한다.(Return문이 여러 개라도 상관없다.)

; 과거에는 하나의 출구(Exit Point)를 가지는 것을 선호하는 구조적인 프로그래밍이 대세였지만, 이렇게 만들기 위해서는 코드가 읽기 복잡해지는 경향이 있다. 따라서, 가능한 읽기 쉬운 코드를 만들기 위해서, "return"문을 여러개 두어도 상관없다. 가능한 이른 복귀(Early Return)을 구현하도록 한다.

```
int function( int x, int y ) {
    int result = 0;

    if (( x > 0 ) && ( y > 0 )) result = x + y; /* 결과를 저장하고 나중에 복귀 한다. */
    if (( x > 0 ) && ( y < 0 )) result = x - y; /* 경우에 따라 추가 연산이 있다. */
    if (( x < 0 ) && ( y > 0 )) result = y - x;
    if (( x < 0 ) && ( y < 0 )) result = -y -x;
    return result;
}

->

int function( int x, int y ) {
    if (( x > 0 ) && ( y > 0 )) return x + y; /* 즉시 복귀 한다. 추가 연산이 없다.*/
    if (( x > 0 ) && ( y < 0 )) return x - y;
    if (( x < 0 ) && ( y > 0 )) return y - x;
    if (( x < 0 ) && ( y < 0 )) return -y -x;
    return 0;
}
```

위의 코드에서는 조건분의 실행에서 “return”을 사용해서 곧바로 원하는 결과값을 돌려주는 방법으로 수정했다. 전자를 좀 더 조건을 나누어 실행 횟수를 줄이는 방법도 가능할 것이다. 하지만, 그런 식으로 구현을 하기 위해서는 중첩된 조건분기 발생하게 되며, 코드의 이해가 어려워진다. 후자는 즉각적인 결과를 돌려주는 방식으로 구현했다. 따라서, 최악의 경우 전자와 비슷한 시간이 걸릴 수도 있다.

Rule 23. 모든 변수는 반드시 사용되기 전에 초기화 되어야 한다.

; 변수(Variable)들은 사용되기 전에 초기화 되어야 한다. 의도하지 않은 결과를 만들지 않기 위해서는 기본적으로(default) 정해진 값을 가지는 것이 좋다. 물론, 사용되지 않는 변수의 경우에는 컴파일러에 의해서 경고 메시지를 주도록 컴파일 옵션을 사용해야 한다. 사용되지 않는 변수는 기본적으로 전부 제거해야 한다. 조건부 컴파일에 따라 사용될 수 있는 경우도 있지만, 이때는 조건을 변수에도 확장해서 만들어 주어야 할 것이다.

```
int x = 0;
int y = 0;
```

```
...
if ( x == y )
{
```

```
do_something();
}
```

Rule 24. 중첩된 "if() else"문은 3단계 이상으로 사용하지 말라.

; 중첩된 조건 분기문은 코드의 복잡도를 높인다. 따라서, 중첩이 많아지면 그만큼 코드를 이해하기 어렵게 만든다. 중첩이 많아지면 실행 경로의 분기도 많아지며, 그것을 테스트하기 위해서 필요한 입력의 조합도 늘어난다. 따라서, 가능한 중첩된 조건 분기문을 사용하는 것을 줄여야 한다. 3단계 이상이 될 경우에는 코드 블록을 다른 함수로 분리하는 것을 고려해보기 바란다. 중첩의 조건을 나눌 수 있거나, 변경할 수 있는 경우에는 코드를 수정하는 것이 올바른 선택이다.

```
if( ... ) {
    if( ... ) {
        if( ... ) {
            ...
        }
        else
        {
            ...
        }
    }
}
....
```

```
if ( ... ) {
    ...
}

if ( ... ) {
    ...
}

} else
{
    ...
}

if( ... ) {
    ...
}
```

코드의 복잡도는 분기문을 만날 때마다 증가하고, "switch()"의 "case()"를 만날 때도 증가한다. 따라서, 분기를 줄이기 위해서 분기가 발생하지 않도록 코드의 구조를 변경하거나, 함수라는 추상화 도구를 사용하는 것이 효과적인 선택이다. 버그는 대부분 코드가 복잡한 상황에서 발생하기에, 복잡한 상황을 최소화하는 것이 좋은 코딩 습관이다.

Rule 25. "switch()"문은 반드시 "default"를 가져야 하며, 각각의 case문도 "break"를 가져야 한다.

; 예외적으로 동일한 동작을 수행해야하는 "case"문의 경우에는 하나의 "break"를 가질 수 있으나, 코드 리뷰에서 반드시 "break"가 있는지를 확인해야 한다. "default"가 없는 경우에는 처리하지 못하는 예상치 못한 입력에 대한 것으로, 생략될 경우에는 언제 오류가 발생할지 알 수 없는 경우도 있다. 따라서,

기본으로 "default"를 항상 쓰는 습관을 길러야 한다.

```
switch(){
    case 0 :
        ...
        break;
    case 1 :
        ...
        break;
    ...
    /* break 없음 */
    default:
        ...
}
```

"switch()"문의 경우, "case"의 증가는 복잡도의 증가를 의미한다. 따라서, 최소한으로 "case"를 유지해야하며, "case" 내에서 실행해야 하는 명령들도 길어선 곤란하다. 이런 경우에는 "case"문에서 처리해야 할 명령들을 묶어서 함수로 만드는 것이 코드를 좀 더 명확하게 읽히도록 만든다. 가능한 "switch()"문을 줄이는 것이 좋으며, 비슷한 "switch()"문들이 코드에 전반적으로 퍼지지 않도록 만들어야 한다. 그런 경우가 생긴다면 코드의 중복이 발생할 가능성이 높다. 이 문제를 해결하려면 리팩토링(Refactoring)과 같은 곳에서 사용하는 추상 팩토리(Abstract Factory)를 활용할 수 있다.

Rule 26. "for()" 루프에서는 종료 조건을 명시하는 값으로 상수 값을 바로 사용해선 안된다. 또한, 루프의 횟수를 제어하는 변수를 루프 내에서 수정해서도 안된다.

; 상수를 그냥 사용하는 것은 코드의 가독성을 떨어뜨리는 주요 요인이다. 따라서, 당연히 상수의 사용은 금지 된다. 또한, 루프가 어떻게 동작할지를 정확히 예상하기 어려운 상황에서, 제어를 담당하는 변수의 값을 변경하는 것은 위험하다. 루프 카운트 변수를 나중에 이용할 필요가 있을 때는, 루프 카운트의 밖에서 선언해 줄 수 있으나, 변수가 두 가지 목적을 가지기에 올바른 선택이 아니다. 각각에 대해서 별도의 변수를 사용하는 것이 규칙이다. 함수가 하나의 역할만 가지듯, 변수도 하나의 역할만 수행해야 한다.

```
int index = 0;
for ( int x = 0; x < MAX_LENGTH; i++ ) { /* C99 Support */
    ...
    index++;
}

for ( int x = 0; x < MAX_LENGTH; i++ ) {
    do_something();
}
```

```
void do_something( void ) {
    for ( int x = 0; x < MAX_LENGTH; i++ ) {
        ...
    }
    return;
}
```

루프는 실행 시간의 대부분 차지하기에, 루프 내에서 함수를 호출하는 것은 좋은 선택이 아니다. 차라리, 루프 전체를 함수로 만드는 것이 올바른 판단이다. 루프 내에서 하나의 명령 라인을 수행한다고 하더라도, 반드시 "{}"을 사용해야 한다. 루프 내부에서 급격하게 제어를 옮기는 경우는 될 수 있으면 없어야 하지만, 필요한 경우에는 사용하던 자원에 대해서 적절한 조치를 취해야 한다. 될 수 있으면 급격하게 실행 경로의 변경은 하지 않는 것이 좋다.

Rule 27. 상수 값과의 비교 시에는 상수를 왼쪽에 사용하라.

; 이는 찾기 어려울 수 있는 버그를 미연에 방지하기 위한 방법으로, "=="을 사용해야 할 때, "="을 실수로 사용해서 발생하는 오류를 제거할 목적으로이다. 이런 경우는 흔히 발생할 수 있으며, 어떤 경우에는 이런 것들을 미연에 방지하도록 편집기에서 “주의” 메시지를 주는 경우도 있다. 관련된 규칙으로 조건문은 항상 “boolean”값을 만들어내는 표현(Expression)만을 사용하는 것이다.

```
#define MAX_LENGTH 1000

if ( x == MAX_LENGTH ) {
    ...
}
->
if ( MAX_LENGTH == x ) {
    ...
}
```

상수에 값을 대입하려고 하면, 컴파일러는 당연히 컴파일 오류를 발생 시킬 것이다. 따라서, 이런 습관을 익히면 사소하지만, 찾기 어려운 버그를 손쉽게 제거할 수 있다. 굳이 변수를 왼쪽에 사용해야 한다면, 코드 리뷰에서 모든 상수의 비교문들이 제대로 되어 있는지 확인해야 할 것이다.

Rule 28. 모든 헤더 파일은 중복 선언이 되지 않도록 하라.

; 헤더 파일들의 중복적인 선언을 회피하기 위한 방법은 조건부 컴파일 옵션과 매크로를 조합해서 다음과 같이 만들어서 사용할 수 있다. 이때, 파일의 이름을 조건으로 명시적으로 사용하도록 한다.

```
#ifndef __XXX_H__
#define __XXX_H__

/* Type definition */
...

#ifndef __cplusplus /* C++에서 사용하기 위해서 - 시작 */
extern "C"{
#endif

/* Function definition */
...

#ifndef __cplusplus /* C++에서 사용하기 위해서 - 끝 */
}
#endif
#endif /* End of __XXX_H__ */
```

이와 같이 만든 헤더 파일을 템플릿(Template)으로 사용해서, 전체 개발자들이 공유해서 사용할 수 있도록 하는 것이 좋다. 마찬가지로 구현 파일에 대한 일반적인 템플릿을 정의한 후에 공유하도록 해야 한

다. 필요하다면, "doxygen"과 같은 프로그램에서 사용하는 주석 형태도 미리 정의된 파일을 템플릿으로 배포해야 할 것이다.

Rule 29. 필요 없는 헤더 파일들은 제거하라.

; 코딩할 때 코드는 정리하면서도, 정작 잠시 필요했다가 사용할 필요가 없어진 헤더 파일들의 "#include"는 제거하지 않는 경우가 많다. 기본적으로 필요 없는 모든 것들은 코드에 남겨두지 말아야 한다. 사용자가 만든 헤더 파일들의 내용도 검토되어야 하며, 순환적인 참조 관계도 제거해 주어야 한다. 헤더 파일을 잘못 사용하는 경우에는 작은 변경에도 매번 전체 코드를 다시 컴파일해야 하는 경우도 발생할 수 있다. 그리고, 헤더 파일이 영향을 주는 범위를 지나치게 크게 가져가면, 작은 수정이 전체 코드에 공통으로 영향을 줄 수 있다.

```
#include "xxx.h"
#include "yyy.h" /* 필요없어진 헤더 파일 */
#include "zzz.h" /* 필요없어진 헤더 파일 */

...
void function_x( void ) {
    ...
}

-->
#include "xxx.h" /* 필요없는 헤더 파일들은 제거 */

...
void function_x( void );
```

헤더 파일을 만드는 원칙은 각각의 구현 파일에 대해서 1개 두는 것이다. 헤더 파일을 두는 곳을 어디로 지정 할 것인지도 중요하다. 즉, 전체 과제의 디렉토리 구조를 만들고, 어디에 헤더 파일을 둘 것인지 결정해야 한다. 헤더 파일은 공용으로 사용하는 경우와, 그렇지 않은 경우로 나눌 수 있으며, 각각의 서브 모듈에서만 사용하는 파일들은 공용 헤더 파일이 있는 디렉토리에 넣지 않아야 한다. 개발의 편의로 모든 헤더 파일을 한 곳에 두는 것은 코드를 나누는 것을 어렵게 만들고, 순환 의존성을 만들 가능성이 높다. 재활용도 당연히 힘들다. 따라서, 각각의 서브 모듈들은 구현 파일들을 두는 디렉토리와 헤더 파일들을 두는 디렉토리로 최소한 2개의 디렉토리를 나누어 만들어야 할 것이다.

Rule 30. 추상화를 위해서 데이터와 함수는 같이 두라.

; 데이터와 함수를 분리된 모듈에 두는 것은 코드의 의존성을 높인다. 따라서, 데이터와 데이터를 다루는 함수는 같이 취급하는 것이 올바른 방법이다. 만약, 상대방이 데이터를 요구한다면, 왜 요구하는지를 보고, 오히려 관련된 일을 직접 처리하고 결과만을 보여주도록 해야한다. 자료구조를 외부에 공개하는 것은 변화의 수용력을 떨어뜨리기에, 될 수 있으면 직접 처리하기보다는 해당 자료구조를 다루는 코드에 의뢰(Request)하는 편을 선택해야 한다.

```
#ifndef __PERSONAL_INFO_H__
#define __PERSONAL_INFO_H__


typedef struct personal_information { /* 헤더 파일에 너무 많은 정보가 있다. */
    char* name;
    unsigned int age;
    char *address;
}

} PERSONAL_INFORMATION;
```

```

/* 구체적인 정보를 숨긴다. */
typedef struct personal_information PERSONAL_INFORMATION;

char *getName( PERSONAL_INFORMATION *pinfo );
unsigned int getAge( PERSONAL_INFORMATION *pinfo );
char *getAddress( PERSONAL_INFORMATION *pinfo );
...
#endif

```

위의 코드에서는 "typedef"를 이용해서 자료구조에 대한 형(type)을 정의했지만, 실제 구현된 상세한 자료구조의 내용까지 보여줄 필요는 없다. 헤더 파일은 외부에 제공할 정보를 알려주는 것과, 자신의 구현을 위한 형(type)을 제공할 목적이면 충분하다. 따라서, 외부에는 최소한의 정보만 보여주도록 정의하면 된다.

여기서 나열한 모든 코딩 룰은 개인적으로 중요하다고 생각해서 선택한 것들 뿐이다. 코딩 룰은 어떤 것을 쓰느냐보다 얼마나 일관성 있게 하는가가 더 중요하다. 과제나 팀에서 선택한 코딩 룰이 있다면, 그것을 전체 과제나 팀에 꾸준히 적용하는 것이 좋다. 주의할 것은 너무 많은 코딩 룰로 인해서 발생하는 비효율을 극복하기 위해서는, 반드시 지켜야 할 것들만 규정으로 만들고 나머지는 특정 응용에 적합한 툴의 지원을 받는 것이 효과적이다.

추가적으로 한가지 더 이야기 한다면, 코딩 스타일에 대해서는 코딩 룰 검사기에서 지원하지 않을 가능성이 높다. 예를 들어, MISRA-C 2012와 같은 것을 지원하는 툴에서는 코딩 스타일이 아닌 버그 가능성을 찾아내는 것에 집중하고 있다. 코딩 룰과 코딩 스타일을 혼동하는 경우가 있는데, 코딩 스타일은 어떻게 코드를 보여줄 것인가에 집중하는 것이고, 코딩 룰은 어떻게 코딩 실수를 방지할 것인가에 초점을 맞추고 있다.

[소스 코드의 관리]

코드는 매일 변한다. 매일 변하는 코드를 관리하는 것은 도구의 도움이 없이는 거의 불가능하다. 누가 어디를 어떻게 고쳤으며, 다른 사람이 개발한 코드를 통합하는 등등의 일은 도구가 반드시 필요하다.

이미 이런 도구들은 소프트웨어 개발 현장에서 널리 사용중이며, 잘 알려진 것으로는 ClearCase, CVS, SVN, Perforce, Git, Bazaar 등이 있다. 회사마다 자신들의 툴 조합에 어울리는 것으로 시스템을 구축하고 있으며, 대부분 비슷한 기능을 제공하고 있다. 개발자들은 작업할 코드를 코드 저장소(Repository)에서 가져와(Check-out) 코드 추가나 수정을 하게되며, 완성된 코드를 저장소로 다시 집어 넣는다(Commit, Check-in). 이를 통해서 코드의 버전이 자동으로 변경되며, 각각의 버전들을 비교해서 달라진 부분을 찾을수도 있다.

```
// 2015년 1월 19일 shkwon 수정 /* 아무 의미가 없다. */
```

누가 언제 어떤 내용을 고쳤는지 확인할 수 있기에, 더 이상 코드 내에 개발자의 이름이나 변경 날짜를 적어두는 것은 무의미한 일이다. 그리고, 코드에 "주석처리된 코드"를 남겨두는 것도 의미가 없다. 보통의 경우 그런 코드들은 참조나 잠시 동안 디버깅 목적으로 남아 있는데, 제대로 정리되지 않은 상태로 남아 코드의 가독성을 해치게 된다. 따라서, 이런 부분들을 다 삭제 되어야 한다.

```
// a = x + y * z; /* 그냥 지워도 별 문제 없다. */
```

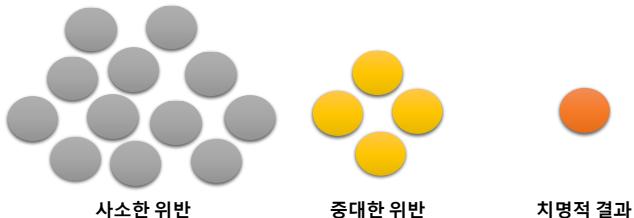
코드를 다른 사람들과 공동으로 개발하는 경우에는 저장소에서 가져올 때와 집어 넣을 때의 시간적인 차이로 인해서 불일치(Conflict, Collision)가 발생하기도 한다. 이때는 직접 사람의 손으로 어떤 코드가

더 최신이며, 혹은 어떻게 넣어야지 통합을 할 수 있는지 툴에 알려주어야 한다. 이때도 변경 전후의 코드와 다른 사람이 작성한 코드의 어떤 부분에서 불일치가 발생했는지 툴로 확인할 수 있으며, 어떤 코드를 사용할지 정해줄 수 있다. 따라서, 개발자들은 소스 코드의 관리에 있어서 다양한 툴을 능수능란하게 다룰 수 있어야 한다.

[컴파일러의 경고는 전부 제거해야 한다.]

컴파일러의 경고를 무시하는 경우를 실제 코드에서 종종 볼 수 있다. 컴파일러의 경고 메시지 수준은 항상 최대로 켜져 있어야 하며, 나오는 모든 경고 메시지는 제거하는 것이 기본이다. 코딩을 잘한다는 것은 이런 "사소한(?)" 부분들에 대해서도 신경을 써야 한다. 물론 지금 당장은 문제가 되지 않는다고 무시할지도 모르지만, 나중에 문제가 될 가능성은 충분하다. 다른 사람이 실수할 가능성을 최소화하는 것도 프로그래머의 책임이다.

하인리히의 법칙



300 : 29 : 1

다른 사람이 그 코드를 가지고 컴파일 했을 때 경고 메시지가 발생한다면, 원래 개발자의 수준을 의심하게 되며 코드에 대해서도 신뢰하지 않는다. 더욱이 비슷한 경고 메시지가 수백개(혹은, 몇 개라도)가 있다면, 그것을 고치는데도 시간을 들여야 하기 때문에, 코드의 원천 개발자를 비난하는게 된다. 다음에 코드를 볼 사람을 위해서 그런 경고들을 수정해야 하기 때문이다.

"gcc -Wall -o helloworld helloworld.c"

컴파일러들은 경고 메시지를 발생시키는 조건을 명시적으로 정할 수 있으며, GCC의 경우에는 "-Wall"과 같이 모든 경고 메시지를 알려달라고 요청할 수 있다. 프로그램의 코드가 만들어지면 처음하는 활동이 컴파일이며, 여기서부터 버그의 가능성을 줄이는 일에 집중해야 한다(물론, 코딩 과정에서 버그 유발을 최소화하는 방법도 중요하지만). 실무에서는 개발자들이 이런 부분에 대해서 크게 의미를 두지 않는 경향이 있는데, 이는 컴파일러의 기능 중 일부 밖에 사용하지 않는 것과 같다. 사람은 항상 실수를 할 수 있는 동물이며, 간단한 실수는 기계적으로 걸러내고, 정말 심각한 오류를 찾아내는 일에 집중해야 한다. 따라서, 컴파일러와 같은 도구를 잘 사용하는 것은 반드시 필요하다.

```
int a = 0; /* if "__DEBUG__" is not defined, then "a" is unused variable */
```

```
#ifdef __DEBUG__
    a = 100;
#endif
```

경고 메시지 중에서 특히 신경쓰지 않는 것들은 "지역 변수의 초기화(Uninitialised)"와 "사용되지 않는 변수(Unused)" 등이 있다. 변수는 사용되기 전에 반드시 초기화를 해주는 습관을 가지고 있어야 하며, 사용되지 않는 변수는 당연히 지워야 한다. 간혹, 특정 모드(예를 들어, 디버깅 시에만 사용하는)에서 사용하는 변수를 제거하지 않아서, 컴파일 옵션을 변경해 주었을 때 생기는 경고도 있다. 그럴 경우에도 당연히 없애는 것이 필요하다.

정적 변수들은 대부분의 경우 "0"으로 자동으로 초기화 되지만, 특정 상황에서는 그것을 기대할 수 없는 경우도 있는데, 지역 변수이건 전역 변수이건 상관없이 초기화 시켜주는 것을 권장한다. 이런 경우는 주로 코드는 수정하지만 변수까지 반영하지 않는 경우에 종종 발생한다. 정당한 이유없이 존재하는 모든 코드는 제거하는 것이 원칙이다.

```
int a = 100;
unsigned int b = a; /* "a" and "b" is not type compatible. */
```

임의로 변수의 타입을 변경하는 것도 문제가 될 수 있다. 그리고, 권장되지 않는 키워드를 이용하는 것도 좋지 않다. 변수는 자신의 타입에 맞게 사용되어야 하며, 호환되지 않는 변수 간에는 문제가 잠재적으로 내포되어 있다. 물론, 그런 문제가 발생하지 않을 가능성도 있지만, 잠재적인 문제는 시간이 흐르면 오류가 될 가능성이 높다. 따라서, 사전에 그런 문제들을 해결하는 것이 도움이 될 것이다.

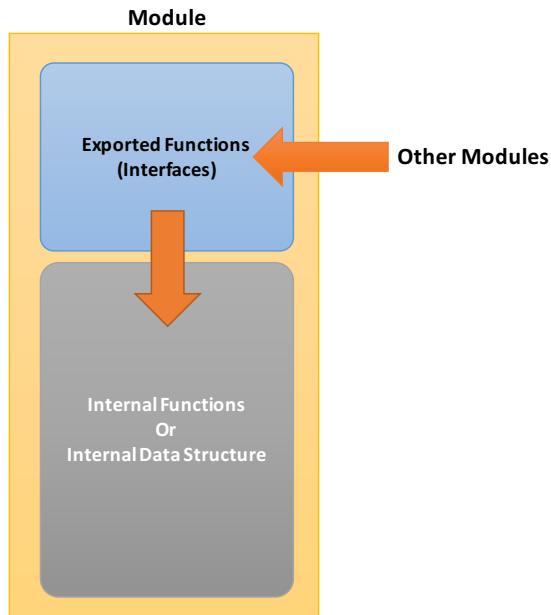
"restrict"나 "auto"같은 키워드들은 별로 권장되지 않는다(이미 낡았다고 표준에서 사용하지 말 것은 권장). 따라서, 그런 키워드(Keyword)를 사용할 필요는 없다. 컴파일러에서 제공하는 경고 메시지를 추적해서 그런 부분들은 과감하게 삭제하도록 해야한다. 코드를 수정할 때는 많은 부분을 한꺼번에 변경하기보다, 작은 부분을 변경하고 더 자주 테스트 하는 것이 효과적이다.

[구조화를 위한 코드 작성]

소프트웨어 개발은 복잡도를 관리하는 것이 핵심이다. 즉, 다양한 수준에서 복잡함을 어떻게 관리할 것인지 정해야 한다. 첫 번째 방법은 외부에서 내부 구조를 볼 수 없도록 숨기는 것이다. 즉, 내부 구조에 접근하는 함수를 만들어, 그것 만을 사용하도록 강제화 하는 것이다. 구조화를 위한 두 번째 방법은 순환 의존 관계를 끊는 것이다. 순환 의존 관계는 서로가 서로를 필요로 하는(의존하는) 관계로, 각각을 다른 모듈로 나눌 수 없도록 만든다. 순환 의존 관계를 끊어내는 방법은 단 방향의 의존 관계로 정리하는 것이다. 순환 의존 관계가 없어지면 모듈과 모듈 간에는 단 방향의 사용 관계가 존재하게 되며, 계층적인 구조를 만드는 기초가 된다.

작성되는 코드는 내부적인 상세한 구현을 외부에 밝혀서는 안된다. 따라서, 자료구조를 직접적으로 사용하려는 행동을 사전에 차단해야 한다. C언어에서는 제공되는 API(Application Programming Interface)에 대한 리스트를 헤더 파일에서 관리하고, 필요한 자료구조의 타입만 허락해야 한다. 실제 자료구조의 상세한 구현은 구현 파일에 있어야 하며, 필드에 접근하는 API를 이용하기 보다는 요청을 하는 형태의 API를 제공해야 한다. 예를 들어, 특정 필드의 값을 얻어오는 API보다, 그 값으로 할 수 있는 일을 API로 제공하는 것이다. 특정한 값을 읽어와야 할 경우가 있다면, 직접 자료구조의 필드를 접근하는 것이 아니라, 값을 얻어오는 함수를 사용해야 할 것이다.

큰 단위의 구조를 작은 단위까지 일관되게 사용해야 한다. 즉, 모듈의 내부적인 상세한 구현에 의존하지 않는 구조를 크게 설계했다면, 내부의 작은 함수들도 그 원칙에 충실히 구현되어야 한다. 될 수 있으면 상세한 구현에 의존하는 것은 가장 말단(하단)의 함수에 한정시켜야 하며, 그 외의 함수들은 추상적인 함수에 의존하는 것이 좋다. 직접적으로 자료를 다루는 것은 가장 낮은 수준의 추상화이며, 그것을 기반으로 상위 함수들이 구성되어야 한다. 자료구조를 직접적으로 접근하는 것을 줄이고, 그런 역할을 하는 함수들의 API를 제공해 관리하도록 한다. 함수들과 변수들은 사용되는 범위를 명확히 해야하며, 파일 수준의 범위(Static)와 그것을 벗어난 범위를 구분해 주어야 할 것이다.



의존성을 최소화하기 위해서 모든 필요한 함수들을 전부 하나의 모듈에 넣는 것은 거대한 하나의 모듈을 만들 뿐이다. 물론, 외적으로 보이게는 완전해 보이겠지만, 내부적인 복잡도는 증가하게 된다. 마찬가지로 복잡함에 대한 관리 실패일 뿐이다. 따라서, 복잡도를 내부적으로 갈무리하는 것이 중요한 일이지만, 그렇다고 내부의 복잡도를 높이기만 해서도 안된다. 내부의 복잡도는 함수가 원칙없이 만들어지기 때문에 발생한다. 함수화의 원칙을 만들고 그 원칙 그대로 코딩하는 것이 좋다. 함수로 만들 수 있는 것들은 모두 함수로 만든다고 생각해야 할 것이다. 가장 쉬운 것은 논리적인 블록들로 나누진 코드의 블록들이다. 그런 부분들을 만난다면, 전부 함수로 바꿀 수 있다고 봐야 한다.

매크로는 가장 간단한 형태를 제외하고(잘못 사용될 가능성이 없는, 예를 들어 최소/최대값 구하기, 특정 상수값의 대체 등과 같은 것을 처리하는 것) 함수로 대체시키는 것이 좋다. 복잡한 조건식도 함수로 만들어야 한다. 반복적으로 실행되는 것들도 모아서 함수로 만들어야 한다. 물론, 이때는 반복 구문도 포함해서 함수가 될 것이다. 함수는 짧아야하고, 한가지 일을 수행해야 하며, 파라미터의 개수가 적어야 한다. 함수의 내부는 일관된 추상화를 가져야 하기에, 특정 함수를 호출하는 코드와 구체적인 일을 하는 코드를 같이 둘 수 없다(같은 추상화 수준이 아니기에). 이런 경우라면, 함수를 호출하는 코드의 연속으로 구성되어야 한다. 구체적인 일은 가장 말단의 함수들이 담당 할 것이다.

순환 의존이 생기는 원인은 하위 모듈이 상위 모듈에 정의된 함수나 값에 의존하기 때문이다. 이때는 상위 모듈에 정의된 함수나 값을 하위 모듈로 옮겨주어야 하며, 상위 모듈에서 해당하는 함수나 값에 대한 호출은 하위 모듈의 구현을 사용할 수 있도록 바꿔주어야 한다. 단 방향 의존 관계로 표현되는 계층화는 하위 계층의 내부 구조 변화가 인터페이스 이상으로 파급되는 것을 막는데 있다. 따라서, 하위 모듈에서 코드의 수정이 발생하더라도, 제공되는 인터페이스가 제공하는 기능은 반드시 기존과 동일하게 동작해야 한다. 이것이 달라진다면 상위 모듈도 그것에 맞게 변경되어야 하기 때문이다.

[변수와 연산자의 적절한 타입]

변수의 타입은 그 변수가 사용 되는 범위 내에서는 다른 타입의 값을 가지지 않아야 한다. 즉, 변수를 정의된 타입과 다르게 사용하면 안된다. 특히, 정수의 경우에는 부호가 있는 정수와 부호가 없는 정수를 섞어서 쓰는 것은 좋지 않다. 될 수 있으면 부호가 없는 정수를 사용하고, 부호가 있는 정수를 다루어야 한다면, 피연산자의 부호는 동일한 것을 사용해야 한다.

예외적으로 포인터의 경우에는 미리 어떻게 사용될지 가정하지 못하는 경우도 있기는 하지만, 될 수 있으면 선언한 타입에 대해서만 사용하는 것이 좋다. 타입 캐스팅을 활용하기 보다는 목적에 맞는 변수를 정의해서 사용하는 것이 항상 우선이다.

```

int *pointer;
int a;

pointer = &a;
*pointer = -1;

unsigned int *pointer2;
unsigned int b;

pointer2 = &b;
*pointer2 = 1;

```

다루어야 할 값이 실제 저장되는 비트(bit)의 수보다 작은 경우에는 정확히 그에 걸맞는 크기를 가지는 타입의 변수를 사용하는 것이 좋지만, 8비트 보다 더 작은 타입의 변수를 사용하기 위해서는 특별한 선언이 필요하다. 또한, 이런 값을 사용한다고 해서 메모리 사용이 줄어들거나(저장 시에는 효과가 있지만)나 성능이 더 좋아지는 것도 아니다. 따라서, 이런 경우에는 저장소에서 가져오거나 다시 저장하는 경우를 제외하고는 가장 일반적인 부호가 없는 정수로 기본 형으로 사용하는 것이 좋다.

사칙 연산에서 발생하는 오류는 변수와 타입을 섞어쓰는 경우에 발생한다. 따라서, 될 수 있으면 변수의 타입이 다른 것을 함께 사용하려고 해선 안된다. 그렇다고 타입 캐스팅(Type Casting)을 계속하는 것도 좋지 않다. 꼭 필요한 경우에는 타입 캐스팅을 해야하지, 대부분의 경우 적절한 타입의 변수를 사용하는 것으로 해결 가능하다.

특히, 코딩 룰을 검사하는 도구는 정수 연산에서 부호있는 정수와 부호 없는 정수를 섞어쓰는 것을 엄격하게 검사하는 경우가 있다. 이는 곳 개발자가 실수할 가능성이 높다는 뜻이기도 한다. 따라서, 변수의 범위를 명확히 하고 정해진 타입을 존속하는 기간동안 변경없이 사용하는 것이 최선이며, 같은 타입의 변수끼리만 연산을 처리하는 것이 좋다.

[변수의 사용 범위를 좁게 만들기]

변수의 사용 범위는 필요한 부분까지만 한정해서 되어야 한다. 예를 들어, 특별한 곳에서만 필요한 변수는 함수 전체 범위에서 접근할 필요가 없다. 변수의 사용 범위를 줄여주는 것은 버그 발생을 좁은 코드내로 한정시킬 수 있다.

```

for( unsigned int i = 0; i < MAX_LENGTH; i ++ ) {
    char *name = "My name is SH Kwon";
    printf("%s\n", name);
}

```

위의 코드와 같이 선언된 변수 “name”은 “for()”문을 벗어나서는 존재하지 않는다. 따라서, “name”에 관련된 문제들도 “for()”문을 벗어날 수 없다. 여기서 보여준 예는 단순한 것이지만, 이를 확장한다면 함수를 짧게 만들고, 관련된 변수들을 정적인 자료구조 보다는 함수 내의 지역 변수로 만들어서 사용하는 것이 버그를 함수내로 묶어 둘 수 있는 방법이라는 것을 알 수 있을 것이다. 따라서, 전역 변수로 선언된 변수에 대한 문제는 전체 시스템의 문제로까지 파악해야 할 필요가 있다.

한가지 주의 할 것은 변수의 범위가 작다고 해서 변수의 이름을 여러 번 반복해서 사용하는 것은 좋지 않다. 변수는 자신의 이름에 맞는 역할을 해야하며, 다른 역할까지 중복해서 가져서는 안된다. 예외적으로 루프를 제어하는 “i, j, k”와 같은 변수들은 일반적으로 “관례상” 재사용을 허락한다. 하지만, 앞의 예와 같이 루프를 벗어나서는 반드시 유효하지 않아야 한다. 일반적으로 사용하는 것들은 그냥 두는 것이 코드를 읽는데 더 익숙하기 때문이다.

[지나치게 큰 지역 변수를 사용하지 않기]

지나치게 큰 지역변수는 스택(Stack) 오버플로우(Overflow)를 발생시킬 가능성이 높다. 특히, 일반적으로 사용하는 운영체제가 없는 상황에서는 시스템이 값자기 이상 동작을 일으키도록 만들수 있다. 원도우와 같은 운영체계를 사용하더라도 특정 크기 이상으로 큰 지역변수를 선언하면, 실행은 될 수 있을지 몰라도 즉시 멈출 것이다. 각각의 운영체제마다 기본적으로 정해진 스택 크기가 다르기 때문에, 특정 운영체제에서 동작한다고 다른 운영체제에서도 같은 것이라고 가정해선 안된다.

```
#define WIDTH 1000
unsigned int result[WIDTH][WIDTH];

for (unsigned int i = 0; i < WIDTH; i++) {
    for (unsigned int j = 0; j < WIDTH; j++) {
        result[i][j] = i + j;
        printf("The result : %d, %d --> %d\n", i, j, result[i][j]);
    }
}
```

위의 코드는 사용하는 운영체제나 컴파일러에 따라 다른 결과를 보여줄 것이다. 어떤 운영체제에서는 실행 중 문제를 일으킬 것이고, 다른 운영체제에서는 제대로 실행될 것이다. 여기서 “result[][]”의 크기가 대략 4Mbytes($=4 \times 1,000 \times 1,000$)로 지역 변수로는 지나치게 크다. 만약 문제가 있다면 지역변수로 선언된 변수를 파일 수준의 전역 변수로 바꿔서 해결할 수도 있을 것이다. 하지만, 그렇게 선언하는 것은 성능 저하를 낳을 것이다. 다른 방법으로는 함수 내부에서 메모리를 할당 받아서 사용할 수도 있다.

[방어적인 코딩]

방어적인 코딩이란 작성한 코드가 어떻게 사용될지 가정하지 말고, 어떤 경우에도 제대로 사용될 수 있도록 만들라는 뜻이다. 또한, 자신이 사용하는 코드(의존하고 있는 코드, 호출하고 있는 코드)들이 제대로 동작할 것이라고 확신하지 말라는 것이다. 따라서, 방어적으로 코딩하기 위해서는 크게 두가지가 필요하다. 작성한 함수의 입력으로 들어오는 값이 가정과 같은지 확인하는 것과, 함수의 호출 후에 반드시 복귀값과 변경될 수 있는 값을 확인 하는 것이다.

```
...
int divide( const int x, const int y ) {
    if( y == 0 ) {
        ERROR_LOG("Cannot do the work!!!\n");
        return 0;
    }
    return (x/y);
}

if (( x == 0 ) || (( result = divide( x, y ) ) == 0 )) {
    return 0;
}
...
```

특히 중요한 것은 대부분의 경우에는 성공하는 함수지만, 간혹 실패할 가능성이 있는 함수를 호출할 때는 주의해야 한다. 특히, 자원에 대한 사용 요청은 언제나 실패할 가능성이 있기에 주의해야 하며(메모리 할당이나, 소켓 생성, 자원 획득에 대한 시스템 호출도 포함), 다른 사람이 만든 코드라고 할지라도 호출 후에는 결과 값을 반드시 확인해야 한다.

코딩은 오류 상황을 어떻게 다룰 것인지 미리 정해야 하며, 오류의 종류와 그 증상에 따라 일관되게 처리해 주어야 한다. 오류를 처리하는 수준(Level)은 “디버그, 정보(로그), 경고, 오류발생, 심각한 오류”와 같은 등급을 매겨서 일관된 처리를 할 수 있는 함수를 정의하고 있어야 할 것이다. 오류는 사용성 개선을 위해서 도움이 되기에, 모아서 관리해주는 것이 좋다.

[함수의 인수 개수를 최소로 유지하기]

함수(Function)을 만든다면 가능한 적은 수의 인수(파라미터)를 가지도록 만드는 것이 좋다. 인수의 수가 늘어나면 테스트해야 할 경우의 수가 늘어난다. 예를 들어, "x, y, z"이라는 인수가 있고 각각이 2가지 값만 가진다면, "2 x 2 x 2"로 총 8가지의 테스트 케이스가 발생한다. 물론, 이것은 극히 테스트 케이스를 단순화 시킨 것이고, "integer"나 "float"등등의 값을 가진다면, 더 많은 경우의 수를 만들어서 테스트 해야한다. 이때도, 모든 값을 다 테스트할 필요는 없고, 몇 가지의 특수한 경우로 한정해서 케이스를 만들수 있다. 어쨌든, 인수가 늘어날수록 함수의 내부구조는 더 복잡해질 것이 분명하다. 인자가 많다는 이야기는 함수 내부에서 외부로 부터 정보 입력을 더 많이 요구한다는 뜻이고, 입력에 대해서 더 많은 일을 해야 한다는 의미로 해석할 수 있다.

```
int doSingleFunction(void) {
    /* Do something here!!! */
    return retail;
}

int doDoubleFunction( int doubleTask ) {
    /* Do something more!!! */
    if ( doubleTask == xyz ) {
        /* Do xyz things */
    } else {
        /* Do not xyz things */
    }
    return retail;
}
```

물론, 인수의 숫자가 적어진다고 해서 함수의 길이가 짧아 질 것이라고 단정할 수는 없다. 하지만, 최소한 더 적은 일을 하거나 한가지 일로 집중될 가능성은 높을 것이다. 인수가 하나도 없다면 함수를 호출해서 그 결과 값만 확인하면 된다. 입력이 없기에 출력은 당연히 한가지 밖에 될 수 없다(물론, 내부의 구현에 따라 복귀값이 달라지겠지만).

극단적으로 인수의 개수를 제한해서 코딩해야 한다면, 선택은 4개 까지만 허용하는 것이 좋다. 하지만, 함수의 길이를 제한할 수 있다면, 100라인 보다 작은 함수를 만들도록 해야한다. 따라서, 대부분의 함수는 4개의 인수를 필요로 하지 않을지도 모른다. 그리고, 자료구조를 외부에 보이지 않게하는 방법을 사용하면, 더 적은 수의 인수를 가지고 함수를 만들 수 있게된다. 가능한 최소의 인수 개수를 가져가는 것을 코딩 규칙으로 정하는 이유도 이해하기 쉬운 코드를 만들기 위함이다. 짧은 라인을 가지는 함수가 이해도 쉽고 테스트도 쉽다는 것은 당연한 이야기다.

문제는 함수의 인자를 줄이기 힘든 경우도 존재할 수 있다는 점이다. 물론, 모든 함수를 100라인 이하로 만드는 것이 좋을지 몰라도, 전혀 상관없는 자료구조들을 하나의 함수에서 다루어야 하는 경우도 분명히 있을 것이다. 하지만, 이런 함수들이 전체 함수의 수에서 차지하는 비율은 극히 적을 것이다(1%미만). 따라서, 원칙적으로 가능한 함수의 인자 수를 줄일 수 있도록 노력해야 할 것이다.

[메모리 할당과 해제]

메모리는 동적으로 할당받고 해제해 줄 수 있는 시스템의 가변적인 자원이다. 대부분의 경우 시스템을 불안정하게 만드는 요인은 가변적인 요소들이 늘어나는 경우이며, 이러한 요소들을 최대한 줄일 수 있다면 시스템을 안정화 시키는데 큰 어려움을 겪지 않을 수 있다. 물론, 반드시 사용해야 한다면, 이런 부분들을 제대로 사용하는 것을 배우는 것이 좋으며, 그렇지 않을 경우에는 길고 지루한 디버깅 과정을 겪는 것을 각오해야 한다.

메모리 할당과 해제를 하기 위해서 사용하는 C의 함수로는 “malloc()”과 “free()”가 대표적이며, 이러한 함수는 운영체제의 도움을 받아서 실행 중인 프로그램의 주소공간(Address Space)에 사용할 수 있는 메모리를 할당받고 해제하게 된다. 이들 두 함수를 사용할 때 유의할 점은 할당과 해제가 일어나는 곳을 가능한 가까운 곳(공간적)에 두거나, 혹은 사용이 끝나면 즉시 해제할 수 있는(시간적) 방법으로 사용해야 한다는 것이다.

메모리 할당과 해제시에는 시스템의 자원 유출(Resource Leak)과, 할당되었다고 생각하지만 실제로는 할당되지 않은 메모리에 대한 접근시 발생하는 오류가 주요 문제점들이다. 이를 해결하기 위해서는 메모리 할당과 해제는 가능한 같은 수준의 함수에서 다루거나, 동일한 함수 내에서 사용되는 것이 좋다. 할당은 한 곳에서 하지만, 해제를 다른 곳에서 하는 것(공간적/시간적으로)은 이런 문제를 악화시킬 가능성인 높다. 또한, 이런 문제가 발생할 경우에는 모든 할당되었지만 해제되지 않고 남아있는 메모리에 대한 조사를 거쳐야 하기 때문에, 문제를 해결하는데도 시간이 오래 걸릴 가능성이 높다.

```
#include <stdio.h>
#include <stdlib.h>

#define ALLOC_SIZE 100

int main(void) {
    char *myMem = NULL;

    myMem = (char*)malloc( sizeof( char ) * ALLOC_SIZE );
    puts("Memory Allocation Test!!!\n");
    free( myMem );

    return EXIT_SUCCESS;
}
```

위의 프로그램은 간단히 "char"타입의 데이터를 가지는 메모리 공간을 할당하고, 해제하는 것을 보여주는 예제다. 이 예제에서 보여주고 싶은 것은 가능한 메모리의 사용이 끝나면 즉각 해제해주라는 점과, 하나의 함수 내에서 될 수 있으면 할당과 해제가 일어나야 한다는 점이다. 물론, 한 쪽에서는 할당을 하고, 다른 곳에서 해제를 하는 것도 가능하다. 그럴 경우 각각이 나누어진 함수라면, 아래와 같이 구현될 수도 있을 것이다.

```
#include <stdio.h>
#include <stdlib.h>

#define ALLOC_SIZE 100

char *memAlloc( int size ) {
    char *tempAllocMem = (char *)malloc( sizeof( char ) * size );
    printf("Allocated : 0x%x\n", (unsigned int)tempAllocMem );
    return tempAllocMem;
```

```
}
```

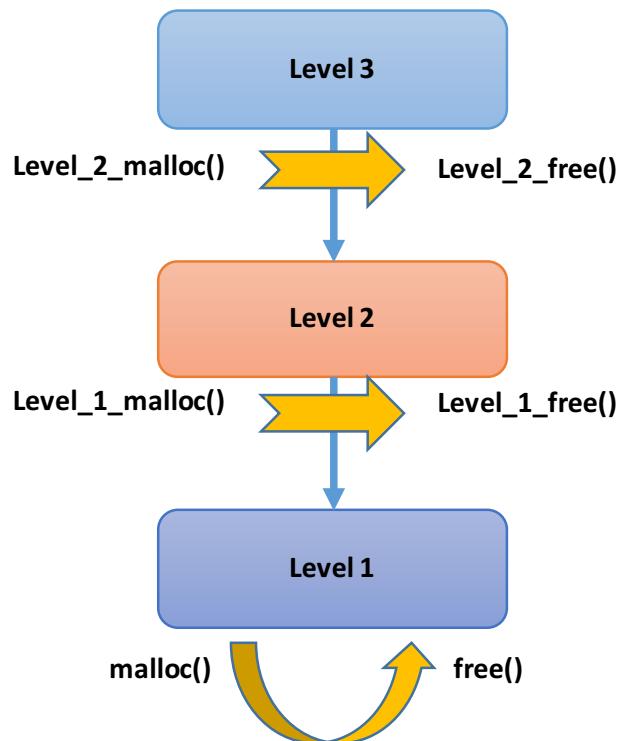
```
void memFree( char *address ) {
    printf("Freed : 0x%x\n", (unsigned int)address );
    free( address );
}

int main(void) {
    char *myMem = NULL;

    puts("Memory Allocation Test!!!\n");
    myMem = memAlloc( sizeof(char) * ALLOC_SIZE );
    snprintf( myMem, sizeof( "Hello, World!!!" ), "Hello, World!!!" );
    printf("%s\n", myMem );
    memFree( myMem );

    return EXIT_SUCCESS;
}
```

즉, 할당과 해제를 담당하는 함수를 각각 같은 수준(다른 함수로 구현되었지만, 하나의 함수에 각각이 사용되었다.)에서 다뤄야 한다는 것이다. 만약, 다른 곳에서 다뤄진다면, 개발자들은 자신이 할당한 메모리에 대해서 해제하지 않을 가능성이 높으며, 이것으로 인해서 자원의 유출이 발생할 가능성이 높다. 만약, 이렇게 만들기 어렵다면, 디버깅을 위해서 시스템에서 제공하는 할당과 해제를 담당하는 함수를 위와 같이 랩퍼(Wrapper)함수로 구현해서 할당과 해제가 어떻게 발생하는지 로그(Log)를 만들어도 될 것이다.



코드를 짤 때 안정적으로 만드는 방법은 미리 메모리를 할당해서 한 곳에서 관리하고, 실행을 마칠 때 그 것을 깨끗하게 정리해주는 방법도 사용할 수 있다. 일종의 메모리 풀(Pool)을 이용한 방법이다. 이것이 좋은 이유는 실행 중에는 메모리가 모자랄 가능성이 없다는 점이다. 문제는 메모리 사용이 늘어나면 사용할 수 없다는 점과 다른 프로그램의 실행에 영향을 받을 가능성이 있다는 점이다. 즉, 메모리 사용이 늘어나면 고정적으로 사용되는 자원을 늘려선 안된다. 하지만, 반드시 실행될 필요가 있을 때는 미리 자

원을 선점하기 위해서 배열과 같은 것을 이용해서 고정적으로 메모리 공간을 확보하는 방법도 사용할 수 있다. 당연히 속도도 더 빠르지만, 활용 안되는 메모리가 늘어날 가능성도 있다.

메모리 할당 자체도 오류가 발생할 가능성이 있기에(시스템에 메모리가 부족한 경우), 제대로 할당 되었는지를 확인해야 한다. 일종의 방어적인 코드를 짜야할 필요가 있다는 것이다. 모든 시스템에 관련된 함수에 대해서는 이런 조치가 필요하다. 즉, 시스템은 우리가 생각한 것 만큼 모든 일을 다 해주는 것은 아니라고 가정해야 할 것이다. 어느 순간 오류가 발생하더라도 안정적으로 실행될 수 있는 프로그램을 만드는 것을 목표로 해야할 것이다.

[위생검사(Sanity Check)하기]

함수를 만들 때, 의도한 값과는 다른 값으로 함수가 호출되는 경우가 있다. 제대로 함수가 호출되고 있는지를 알기 위해서는 일종의 "위생 검사(Sanity Check)"와 같은 것이 필요하다. 물론, 나중에 양산에 들어가는 코드에서는 분리되어야 하기에 "#if __DEBUG__ ~ #endif"와 같은 조건부 컴파일을 간단히 사용할 수 있다. 헤더 파일과 같은 곳에 간단히 매크로를 정의해서 처리하거나, 혹은 미리 제공되는 "assert()"와 같은 것을 사용해도 된다. 이 때 주의할 것은 시간에 민감한 코드를 만들 경우에는 "printf()"와 같은 출력문도 영향을 줄 수 있기 때문에 주의해서 사용해야 한다는 점이다. 시간에 민감한 코드를 만드는 것 자체가 좋은 선택은 아니지만, 어쩔 수 없는 경우도 있기에 주의해야 한다.

```
void function( int parameter1, int parameter2 ) {
#ifndef __DEBUG__
    if (( parameter1 > MAX_PARAM_INPUT )||( parameter2 < MIN_PARAM_INPUT )) {
        PRINT_LOG("Input Parameter Range Error!!\n");
        return;
    }
#endif
    ...
    return;
}
```

위의 함수는 간단히 입력을 받는 파라미터의 값이 특정 범위를 만족 하는지를 검사하는 코드를 조건부 컴파일을 통해서 넣어둔 경우다. 물론, 조건부 컴파일이 마음에 들지 않는다면, 사용할 수 있는 매크로를 따로 정의해서 일관되게 전체 코드에서 사용해도 된다. 중요한 점은 항상 함수가 제대로 사용될 것이라는 가정을 하지 말라는 것이다. 특히, 디버깅 시에는 이런 부분을 잘 활용할 수 있어야 한다.

[매크로를 활용한 디버깅(Debugging) 메시지의 출력]

디버그(Debug)시에 프린트 문을 이용한 출력을 보는 경우가 많다. 물론, 만들고자 하는 프로그램이 속도에 민감한 경우에는 이런 디버그 출력문 자체가 성능이나 동작에 영향을 주기 때문에 주의해서 사용해야 하지만, 일반적으로 가장 손쉽게 프로그램의 동작을 알기 위해서 사용할 수 있는 방법이다.

```
#ifdef __DEBUG__
    #define DEBUG_PRINT( format, args... ) printf( "<%s, %d, %s >" format, __FILE__,
__LINE__, __func__, ##args )
#else
    #define DEBUG_PRINT( format, args... )
#endif
```

위에서 정의한 매크로는 "__DEBUG__"이 정의(Define)되었다면, "DEBUG_PRINT()"를 이용해서, 파일/라인/함수의 이름을 순서대로 출력하고, 원하는 문자열을 "printf()"를 이용해서 출력하도록 만든다. 만약, "__DEBUG__"이 정의되지 않았다면 아무런 일도 하지 않을 것이다.

```
#include <stdio.h>
#include <stdlib.h>

#define __DEBUG__

#ifndef __DEBUG__
    #define DEBUG_PRINT( format, args... ) printf( "<%s, %d, %s >" format, __FILE__,
__LINE__, __func__, ##args )
#else
    #define DEBUG_PRINT( format, args... )
#endif

int main(void) {
    puts("!!!This is Debug Macro Print Program!!!");
    /* Print Debug Message */
    DEBUG_PRINT( "Hello, World!!!\n");

    return EXIT_SUCCESS;
}
```

위의 프로그램에서는 먼저 "#define" 이용해서 "__DEBUG__"를 정의한 후에, "main()" 함수에서 디버그 메시지를 출력한 경우다. 아래는 그 출력을 결과를 보여준다.

<..\\src\\DebugMacro.c, 25, main >Hello, World!!!

[모듈별 디버깅 매크로의 정의]

코딩을 여러 사람이 하다보면, 자기가 보고 싶은 디버그 메시지를 찾기가 어려운 경우가 있다. 이때는 각각의 모듈별로 디버그 메시지를 켜고 끌 수 있도록 할 수 있어야 한다. 따라서, 이를 위해서는 디버그 메세지만 담당하는 매크로를 정의할 수 있는 헤더 파일이 필요하다. 예를 들어, 이 파일을 "debug.h"라고 한다면 다음과 같이 정의할 수 있을 것이다.

```
#ifndef __debug_h__
#define __debug_h__

#ifndef __DEBUG
#ifndef __MY_MODULE_DEBUG__
    my_debug_print( format, args... ) printf( "[ %s, %d : %s ]" format, __FILE__,
__LINE__, __func__, ##args )
#else
    my_debug_print( format, args... )
#endif /* End of My Module Debug
#ifndef __OTHER_MODULE_DEBUG__
    ...
#endif /* ... */
#else /* Else of Debug */
    my_debug_print( format, args... )
    ...
#endif /* End of Debug */
#endif /* End of File */
```

위의 코드는 "__DEBUG__"이 정의된 경우, 자신의 코드와 관련된 모듈들만 디버그 메시지를 켜도록 해준다. 다른 모듈에서 출력되는 디버그 메시지와 섞이지 않도록 만들기 위해서, 각각의 모듈에 해당하는 부분만 "#define"으로 처리해 주면 될 것이다. 컴파일시나 코드에서 자신의 모듈에 해당하는 "#define"을 해주면, 원하는 모듈의 디버그 메시지만 켜고 끌 수 있게 된다.

[안전한 코딩(Secure Coding)에 대해]

보안에 신경을 써야한다는 것은 다들 알지만, 막상 그것을 코드에 반영하기 위한 방법은 낯선 것이 사실이다. 이런 것들을 다루는 분야가 "Secure Coding"이라고 하며, 이미 상당 부분 언어적으로 취약한 부분에 대한 대비책이 있다. 여기서는 간단한 예를 통해서 우리가 알고 있는 함수도 문제가 생길 수 있다는 점을 보고자 한다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct stringBuffer {
    char buffer[ 10 ];
    char helloWorld[ 16 ];
} MyStrings;

int main(void) {
    puts("!!!Secure Coding Example!!!");

    strncpy( MyStrings.helloWorld, "Hello, World!!!", 16 );
    scanf("%s", MyStrings.buffer);
    printf("%s\n", MyStrings.buffer);
    printf("%s\n", MyStrings.helloWorld);

    return EXIT_SUCCESS;
}
```

위의 코드는 하나의 구조체에 두개의 버퍼로 선언된 필드를 두고, 각각을 다른 목적으로 사용하고 있는 경우다. "scanf()"함수를 이용해서 자료를 읽어오는 경우, 자료구조의 크기와는 무관하게 데이터를 읽어올 수 있으며, 이의 결과로 원했던 결과를 만들어내지 못하는 경우가 발생할 수 있다. "scanf()"를 통해서 읽어온 데이터는 "buffer"에 저장했고, "Hello, World!!!"라는 문자열을 이어진 "helloWorld"에 저장했지만, "scanf()"함수가 "buffer"의 크기보다 더 큰 문자열을 저장해서, 이전에 사용하려고 저장한 데이터를 침범하게 된다. 출력은 아래와 같다.

(입력 문자열->) dfadsfadfadfasdfasdfa

!!!Secure Coding Example!!!

dfadsfadfadfasdfasdfa (<- "buffer"에 저장된 문자열)

dfadsfadfadfasdfasdfa (<- "helloWorld"에 저장된 문자열)

출력에서 보듯이 입력된 값이, "buffer'를 넘어서 다른 메모리 공간까지 넘어가 버렸다. 만약, 코드가 더 복잡했다면, 영뚱한 메모리 공간을 침범했거나, 혹은 스택(Stack)과 같은 공간을 침범해서 프로그램에 심각한 보안 구멍을 만들수도 있었을 것이다. 여기서는 단지 간단한 예를 통해서 이런 문제가 발생할 수 있다는 것만 보여주었다. 관심이 있다면 “CERT”에서 제공하는 코딩 표준 가이드를 읽어볼 것을 권한다.

[스트링(String) 관련 함수를 사용할 때 주의할 점]

스트링 함수들은 문자열을 조작할 때 사용한다. 사용시 주의할 점은 크기를 명확히 해야한다는 점이다. 스트링은 문자열로 구성된 것으로 스트링의 끝을 나타내기 위해서 "\0"를 사용하고 있다. 스트링을 복사하거나 조작할 때 "\0"에 대한 처리를 잘못하면, 문자열 연산은 "\0"을 만날 때까지 적용될 가능성이 있다. 의도하지 않은 메모리 공간을 침범할 가능성이 있으므로, 최대한 문자열의 크기를 명확히 나타내는 함수들을 사용하는 것이 버그를 줄여줄 것이다.

예를 들어, "strcpy()" 함수 대신에 "strncpy()" 함수를 사용하는 것이 명확히 몇 개의 문자를 복사할 것인지를 나타낼 수 있기에, 조작을 실수할 가능성이 적다. 물론, 문자열을 조작한 후에는 반드시 "\0"에 대한 것을 주의해야 한다. 문자열의 끝을 나타내는 부분이 없으면, 그것에 의존한 다른 코드들은 문제를 발생 시킬 가능성이 있기 때문이다. 따라서, 항상 사용하고자 하는 문자열보다 하나 더 큰 문자열 저장공간을 확보해서, 마지막에는 "\0"이 문자열의 끝을 나타낼 수 있도록 해주어야 한다.

[경계조건에 대한 주의]

코드는 경계값을 처리하는 과정에서 많은 문제를 일으킨다. 배열의 "[0]"과 "[MAX_ARRAY_SIZE]", 정수가 가질 수 있는 최대 혹은 최소값, 그 값을 넘어서는 순간 부호의 변경이나 오버플로우의 발생, 해제된 메모리 공간에 대한 포인터를 이용한 참조, 조건문의 조건이 참이나 거짓에서 반대로 변하는 순간, 문자열의 끝을 나타내는 "\0"값의 유무 등등 다양한 오류들이 경계 조건을 제대로 검사하지 않아서 발생 한다. 따라서, 코드의 신뢰를 높이기 위해서는 경계가 되는 부분을 찾아서 적절한 처리를 해주어야 한다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_ARRAY_SIZE 100

int main(void) {
    puts("Boundary Condition Example 01");

    unsigned int array[MAX_ARRAY_SIZE];
    array[MAX_ARRAY_SIZE] = 0; /* array overflow */

    printf("The size : %lu\n", sizeof("Hello, World!!!"));
    char* string = (char*) malloc(sizeof("Hello, World!!!"));
    printf("The size : %lu\n", sizeof("Hello, World!!!"));
    strcpy(string, "Hello, World!!!"); /* string copy to smaller memory space */

    return EXIT_SUCCESS;
}
```

위의 코드는 컴파일에서 경고 메시지가 보이지만 실행하는데 문제가 발생하지 않을 수도 있다. 이런 오류들은 다른 코드를 실행하면서 오류를 유발할 수 있기에, 찾아내기 힘들 가능성이 높다. 따라서, 오류가 발생했을 때는 즉시 알 수 있는 방법을 미리 정의해 두고 사용하는 것이 좋다. 배열의 인덱스 오류는 컴파일러나 정적 분석 툴이 찾아줄 가능성이 높기에, 경고 메시지를 확인하고 처리하면 된다. 두 번째 오류의 경우에는 문자열 복제 전후에 크기를 확인하는 방법을 사용하거나, 사용하려는 문자열을 실수로 잘못 적지 않았는지 크기를 확인해야 할 것이다.

특히, C언어의 경우 컴파일러가 잘못된 사용에 대해서 사용자의 의도로 보는 경우가 많기 때문에, 오류가 아닌 것으로 생각할 수도 있다. 물론, 지금 당장은 그런 조건이 만족되지 않을 수도 있고, 제한된 입력으로 인해서 완전히 코드의 속속들이까지 검증이 되지 않았을 수도 있다. 하지만, 그렇다고 버그가 발생

하지 않을 것이고 가정하지 못한다. 미래에 어느 시점에 코드의 변경이 발생한다면 버그 될 가능성은 남은 것이다. 따라서, 차라리 그 시간까지 기다리기 보다 발견하면 즉시 고치는 것이 좋다.

[3차원 이상의 배열 사용하지 않기]

배열은 다차원으로 만들 수 있다. 하지만, 지나지게 차원이 많아지면 이해하기 어려운 코드를 만든다. 따라서, 차원을 제한해서 사용하는 것이 좋다. 다음은 3차원 이상의 배열을 사용하는 경우를 보여 준다.

```
int Array[10][10][10];
```

이를 다루기 위해서는 제어문에 "for()", "while()" 등의 반복문을 사용해야 하며, 메모리의 구성이 선형적이기에 사용하는 CPU의 데이터를 위한 캐시를 넘어설 가능성도 높다(물론, 앞의 예는 넘어설 가능성이 적지만). 따라서, 만약 이런 경우를 만난다면 정말 그렇게 사용해야 하는지 고민해보는 것이 좋다. 차라리, 3차원 이상의 배열이 필요한 경우 2차원 배열과 포인터를 이용해서 정의하고, 3차원 배열 이상에 대해서 포인터를 이용한 접근이 코드를 이해하기 쉽게 만들어 줄 것이다. 또한, 반복문 자체도 2차원 반복문에 대해서는 중첩을 2단계까지 할 가능성이 높다.

```
int *Array[10][10];
int OtherArray[1000];
int *Temp;

for ( int i = 0; i < 10; i++ ) {
    for ( int j = 0; j < 10; j++ ) {
        Array[ i ][ j ] = &OtherArray[ i * 100 + j * 10 ];
    }
}
...
Temp = Array[ x, y ];
for ( int i = 0; i < 10; i++ ) {
    *(Temp + i) = xxx;
}
...
```

위의 코드는 1000개로 선언된 배열을 10개씩 나누어, 각각을 2차원 배열에 시작 주소만 저장한 경우다. 초기화는 한 번만 실행 되며, 각각의 원소에 대한 접근은 10개씩 나누어서 저장된 곳을 직접 접근해서 처리하게 된다. 즉, 원소를 다룰 때는 지역적으로 연속된 10개의 공간만 다루면 될 것이다. 이것이 어색할 수도 있지만, 나중에 지역적인 코드를 읽을 때는 도움이 될 수 있다.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *Array2D[10][10];
    int Array3D[1000];
    /* Make 3D Array */
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            Array2D[i][j] = &Array3D[i * 100 + j * 10];
        }
    }
    /* Initialize 3D Array */
```

```

for (int i = 0; i < 1000; i++) {
    Array3D[i] = i;
}

/* Check 3D Array Through 2D Array */
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        int *temp = Array2D[i][j];
        for (int k = 0; k < 10; k++) {
            printf("%4d", *(temp + k));
            if ((i * 100 + j * 10 + k) % 50 == 49) {
                printf("\n");
            }
        }
    }
}
return EXIT_SUCCESS;
}

```

대부분의 문제는 2차원 배열 이상은 필요하지 않은 경우가 많다. 사람은 차원이 많아지면 그만큼 머리속에 그리는 이미지를 만들기 어려워한다. 선이나 평면에 대해서는 익숙해 하지만, 3차원 이상이 되면, 상상하는데 어려움을 겪는다. 따라서, 문제를 푸는 것도 중요하지만, 풀이가 된 문제를 제대로 다른 사람에게 쉽게 이해가 가도록 만드는 것이 프로그래밍의 핵심이라는 점을 기억해야 한다.

```

#include <stdio.h>
#include <stdlib.h>

int M = 10, N = 10;

void make_3D_array(int* array2D[][N], int x, int y, int array3D[]) { /* 간단하다 */
    for (int i = 0; i < x; i++) {
        for (int j = 0; j < y; j++) {
            array2D[i][j] = &array3D[i * 100 + j * 10];
        }
    }
}

void change_3D_array(int array3D[]) { /* 간단하다. */
    for (int i = 0; i < 1000; i++) {
        array3D[i] = i;
    }
}

void print_2D_array(int *array2D[][N], int x, int y) { /* 너무 복잡하다. */
    for (int i = 0; i < x; i++) {
        for (int j = 0; j < y; j++) {
            int *temp = array2D[i][j];
            for (int k = 0; k < 10; k++) {
                printf("%4d", *(temp + k));
                if ((i * 100 + j * 10 + k) % 25 == 24) {
                    printf("\n");
                }
            }
        }
    }
}

```

```

        }
    }
}
}

int main(void) {
    puts("!!!MultiDimensional Array Test 03!!!");
    int *Array2D[M][N];
    int Array3D[1000];

    make_3D_array(Array2D, 10, 10, Array3D);
    change_3D_array(Array3D);
    print_2D_array(Array2D, 10, 10);

    return EXIT_SUCCESS;
}

```

위의 코드는 앞에서 보여준 코드를 2차원 배열을 함수로 전달해서 처리하는 것을 보여 준다. 1차원 배열의 경우에는 “배열”이라는 것만 함수가 알아야 하지만, 2차원 배열 부터 길이에 대한 정보를 알려줄 수 있어야 한다. 호출하는 측에서는 배열의 이름만 전달하고, 호출되는 함수는 원소의 개수를 정보로 넘겨 받아야 한다. 코드에서는 넘겨 받을 정보의 일부를 배열의 크기를 결정하기 위해서 먼저 선언해서 전달 했다. 물론, 배열 내부에 특별한 값을 통해서 마지막 값을 나타내는 방법을 사용한다면, 배열의 크기를 전달하지 않아도 된다. 하지만, 이때는 반드시 사용할 가능성이 전혀 없는 값으로 배열의 마지막을 나타내야 할 것이다.

[포인터 연산도 연산자의 우선 순위를 보이게 만들어라.]

연산자의 우선 순위를 일일이 기억하기는 힘들다. 간혹 잘못 사용 하기라도 하면, 어디서 문제가 발생했는지를 찾는 것도 어렵다. 특히, 포인터와 관련된 연산자는 사용하는 것이 더 까다롭고 실수할 가능성도 높다. 이럴 바에는 그냥 명확하게 우선 순위를 "()를 이용해서 표현해주는 것이 코드를 이해하는데 훨씬 도움이 될 것이다.

```

char *p = "Hello, World";

for ( int i=0; i<sizeof("Hello, World"); i++ ) {
    printf("%c", *p + i );
    printf("%c", *(p + i));
}

```

이 정도 코드는 그나마 쉬운 편이지만, “**++**”나 “**—**”가 포인터 변수의 앞 뒤에 놓이면 생각보다 읽기 어려운 코드가 된다. 예를 들어, 아래의 코드는 생각보다 이해가 쉽지 않을 것이다.

(p***++**) or (**(*p***++**)*), (**—*p***) or (**—(**p*)**)**

모호하게 보이는 코드보다, 명확하게 보이는 코드를 만드는 것이 좋다. 문법이 허락한다고 모든 코드가 이해하기 쉬운 코드가 되는 것은 아니다. 코드는 작성한 사람의 편의로 만들어지는 것이 아니라, 읽는 사람에게 의도를 정확히 전달할 목적으로 작성되어야 한다.

(*(*p*++**)), (*(*—*p*)*)**

"+, -, *, /"등도 마찬가지로 의도를 명확히 표현하도록 작성되어야 한다. 구조체를 포인터를 이용해서 필드에 접근할 때도, 마찬가지로 명확한 의도를 나타내는 방식으로 코딩하는 것이 좋다.

(*p).name or p->name

위의 코드에서 보듯이, “.”을 사용하는 것이 명확한 의도를 표현하는데 어색하게 보인다(개인적인 차이겠지만). 따라서, 이때는 "->"를 사용하는 것이 더 좋을 것이다. 만약, "++, --"와 같은 것이 추가되어야 할 경우라면, 필드를 접근하는 것과 포인터를 연산하는 것을 분리해서 할 수 있다. 짧게 코드를 만드는 것도 중요하지만, 코드를 읽는 사람의 입장에서라면 풀어서 만들어준 코드가 가독성이 더 높을 것이다.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    puts("!!!Operator Precedence!!!");
    /* Compare pointer operator precedence */
    char *p = "Hello, World!!!";

    for (int i = 0; i < sizeof("Hello, World!!"); i++) {
        printf("%c, %c", *p + i, *(p + i)); /* 흔히 발생하기 쉬운 실수 */
        printf("\n");
    }
    return EXIT_SUCCESS;
}
```

위의 코드를 실행하면, 출력 결과가 다르게 나온다는 것을 볼 수 있다. 코드를 읽는 것이 불편하면 실수도 많이 발생하며, 결과적으로 디버깅 시간도 길어진다. 사소한 선택이지만 쌓이면 결과가 달라진다. 포인터가 C언어가 주는 혜택이라면, 그것을 잘 쓰기 위한 방법도 제대로 이해하고 있어야 할 것이다.

[“for()” 반복문의 사용법]

“for()”루프를 사용하는 방법은 단순하지만, 프로그래머들은 최대한 언어가 제공하는 문법을 이용해서 부가적인 일까지 처리하려는 경향이 있다. 예를 들어, “for()”루프를 제어 하는 변수의 초기화나 증감에 대해 연산을 추가하는 경우가 있다. 다음의 코드를 보도록 하자.

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_ARRAY_SIZE 100

void function(void) {
    unsigned int array[MAX_ARRAY_SIZE] = { 0 };
    unsigned int i = 1;
    unsigned int j = 10;

    for (; (i < MAX_ARRAY_SIZE) && (j < MAX_ARRAY_SIZE / 2); i++, j++) {
        array[i] = j;
    }

    for (unsigned int k = 0; k < MAX_ARRAY_SIZE; k++) {
        printf("The array value[ %d ] : %d\n", k, array[k]);
    }
}
```

```
    return;
}
```

위의 함수는 "for()"루프의 제어변수 값을 외부에서 읽어 왔으며, 조건 검사에서 두개의 변수를 이용하고 있다. 마지막으로 제어 변수 값의 변경에 대해서 두 개의 변수 각각을 별도로 증가시켜 주고 있다. 여기서 사용된 "for()"루프 정도는 충분히 이해할 수 있을 것이다. 하지만, 그 출력을 위해서 사용한 "for()"루프에서 알 수 있듯이, 일반적으로 사용하는 "for()"루프를 사용하는 것이 코드를 읽는 사람 입장에서 훨씬 편하다. 이미 많이 보았던 것이기에 별다른 노력없이 실제로 하는 일에 집중할 수 있기 때문이다. 또한, 제어 목적으로 사용하는 변수의 갯수가 늘어나기에 추가적인 변경에 대해서 영향을 받을 가능성도 같이 늘어나게 된다.

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_ARRAY_SIZE 100

void function(void) {
    unsigned int array[MAX_ARRAY_SIZE] = { 0 };
    unsigned int j = 10;

    for (unsigned int i = 1; i < MAX_ARRAY_SIZE; i++) {
        array[i] = j;
        if (j >= (MAX_ARRAY_SIZE / 2)) {
            break;
        }
        j++;
    }

    for (unsigned int k = 0; k < MAX_ARRAY_SIZE; k++) {
        printf("The array value[ %d ] : %d\n", k, array[k]);
    }

    return;
}
```

위의 함수는 앞에서 만들었던 함수의 "for()"루프를 개선한 것이다. 루프 내의 코드는 이전의 코드에 비해서 늘어났지만, "for()"루프 제어에 대한 부분은 익숙하게 알고 있는 형태로 바뀌었다. 내부에 "if()"조건문을 통해서 루프가 갑자기 끝날 수 있다는 것만 제외하면, 큰 부담은 없이 사용할 수 있게 되었다. 또한, 제어 변수 역할을 하던 "j"도 따로 관리하도록 만들었다. 따라서, "for()"루프를 위한 주요 제어 변수와 제어를 옮기는 변수가 각각 따로 변경할 수 있게 되었다(한가지 덧붙이면, 일반적인 "for()"루프의 제어 변수는 "0"부터 시작하는 것은 관례).

앞의 두 개의 코드 중에서 어떤 것이 더 이해가 쉬운지는 개인의 선호에 따라 다르겠지만, 될 수 있으면 복잡함을 풀어내는 방법으로 코드를 작성하는 것이 좋다.

[열거형(Enumerator)을 이용한 상수값(Magic Number)의 대체]

상수값을 직접 코드에 쓰는 것을 "하드 코딩(Hard Coding)"이라고 한다. 이런 식의 코딩은 변경에 취약하며, 값에 대한 설명이 없기에 코드를 읽는 사람에게 부담을 주게된다. 물론, 주석으로 주면 상관없다고

이야기 할지도 모르지만, 주석보다 더 나은 방법이 있다면 그것을 사용하는 것이 좋다. 또한, 모든 값을 추적하면서 일일이 주석을 다는 일도 쉬운 일은 아니다. 대신 상수값에 대한 이름을 주는 것이 현명하다.

```
const int max_size = 100;
```

위의 같이 상수에 대한 값은 적절한 상수 변수로 표현 될 수 있다. 군집의 성격을 가진 상수들에 대해서도 마찬가지로 열거형(Enumerator)을 사용해서 처리할 수 있다. 물론, 이때는 변수가 아닌 값으로 바뀐다는 점이 앞의 코드와 다르다. 참고로 변수인 경우에는 변수의 이름을 디버거에서 사용할 수 있다는 장점도 있다.

```
enum VALUE {
    xxx = 0,
    yyy = 1,
    zzz = 2,
    ...
    END /* 마지막 값이라는 것을 표시 */
};
```

이때, 내부적으로 몇 개나 값이 존재하는지를 알기 위해서, 마지막에 특별한 값을 적어줄 수 있다. 이런 식으로 표현하면, 각각의 군집된 값 들에 대한 포괄적인 이해를 높임과 동시에, 한정된 범위를 벗어나는 경우 오류를 쉽게 찾아낼 수도 있다. 컴파일러가 그런 역할을 대신해 줄 것이다.

```
if ( x == 256 ) { /* 256 is Size of specific value */
    ...
}
-->
const int SIZE_OF_VALUE = 256;
...
if ( x == SIZE_OF_VALUE ) {
    ...
}
```

상수에 이름을 붙여주는 것은 디버깅에도 도움을 준다. 그냥 상수값이 등장하면, 그 상수의 의미를 파악하기 힘들지만, 이름이 붙은 상수는 그 자체로 코드의 이해를 돋는다. 물론, "read-only" 메모리를 차지하는 오버헤드가 있지만, 프로그램에서 이해가 되지 않는 상수 갯수가 늘어나는 것 보다, 이해되는 코드를 만드는 것이 더 가치가 있을 것이다.

물론, 이런 자그마한 크기의 램 공간 마저도 부족하다고 이야기하는 경우도 있다. 그런 상황이라면, 오히려 코드의 중복을 찾아서 없애는 것이 더 바람직할 수도 있다. 생각보다 코드의 중복은 자주 있으며, 도구를 이용해서 중복된 코드를 찾는 일은 어려운 일이 아니다. 중복된 코드는 함수의 대상이 되며, 함수 호출의 오버헤드는 큰 비용이 아니다.

```
#define MAX_LENGTH 256
int global_variable;

void function( void ) {
    for ( int i = 0; i < MAX_LENGTH; i++ ) {
        ...
        global_variable += i;
    }
}
```

```

        return;
    }
-->
#define MAX_LENGTH 256
int global_variable; /* 가능한 전역변수 사용을 줄여야 하지만... */

void function( void ) {
    int local_variable = global_variable;

    for( int i = 0; i < MAX_LENGTH; i++ ) { /* 루프내에서 전역변수 접근을 없앴다. */
        local_variable += i;
    }

    ...
    global_variable = local_variable;

    return;
}

```

위의 코드는 루프에서 주로 상수 값을 사용하는 경우를 예로 보여 준다. 추가적으로 전역변수에 대한 잡은 접근을 하지 않기 위해서, 지역변수를 이용한 계산을 사용하고 있다. 컴파일러가 루프에 대한 최적화를 더 잘 할 수 있도록 힌트를 주기 위함이다.

램의 크기에 민감한 수준의 코드라면 정말 적은 자원을 가지고 있는 기기에 들어가는 코드일 것이다. 그런 코드라면 성능도 중요한 품질 요소이며, 단가(Cost)도 중요한 고려사항이다. 하지만, 그렇다고 코드의 이해를 돋지않는 코딩이 정당화 되는 것은 아니다. 왜냐하면, 최적화를 위해서는 어쨌든 그 코드를 이해해야 하기 때문이다. 더 잘 이해하면 할수록 더 나은 최적화를 할 수 있기 때문이다.

```

enum STATE {
    NONE,
    ...
    READY = 0x0001
    STOP = 0x0002
    RUNNING = 0x0004
    ...
};

if ( x | ( READY | RUNNING ) ) {
    ...
}

```

마지막으로, 열거형의 첫 번째가 0이란 값이 자동으로 주어진다는 사실에 기초해서, 그 값을 "NONE"과 같이 특수한 목적으로 사용하는 것도 좋다. 즉, 해당하는 타입에 대해서 설정 되지 않았다는 것으로 사용할 수 있다. 또한, 각각의 값에 대해서 "Bit"연산을 할 수 있도록 ("&, |"), 값을 설정하는데도 활용할 수 있다.

[이름 만들기]

변수(Variable)나 매크로(Macro)들의 이름은 너무 짧아도 안되지만, 너무 길어도 문제가 될 수 있다. 특정 컴파일러에서는 "31"자까지만 의미를 지닌다고 판단할 수 있기에, 그 길이 이상의 이름을 주는 것은 프로그램의 호환성에 문제가 될 수도 있다. 하지만, 그렇다고 이름의 길이를 임의로 짧게 약자로 주는 것도 별로 도움이 되지 않는다. "이름은 충분히 존재 이유를 설명할 수 있어야 한다."는 것이 원칙이다. 흔

히 코딩을 처음하는 사람들이 실수하는 부분이 이름을 제대로 만들지 않는다는 것이다. 이름은 프로그램을 코드 수준에서 이해하는데 결정적인 역할을 하며, 일반적으로 값(Value)을 대체해서 사용하기에, 그 값이 의미하는 바를 표현한다고 볼 수 있다.

```

int a;                                /* 의미없는 이름이다. */
unsigned int age;                      /* 무슨 의미인지는 더 코드를 읽어야 알 수 있다. */
#define BAUDRATE 19600                 /* 전문 용어 */
#define MAX( x, y ) ((x) > (y) ? (x) : (y)) /* 일반적으로 사용하는 표현 */
#define NET_BAUDRATE 25600              /* 모듈 이름을 포함하는 상수값 정의 */

void swap_two_name( char *first_name, char *second_name ); /* 무슨 일을 하는지 묘사 */
/* 모듈을 명시한 묘사적인 이름 */
void net_swap_two_address( char *first_ip_address, char *second_ip_address );

```

의미를 전달한다는 점에서 단순히 알파벳을 사용하는 것보다, "묘사적인 이름"을 주는 것이 좋다. 주로 이름을 줄 때, 변수의 이름은 "명사"를 사용하고, 매크로의 경우에는 "동사+[명사]"로 하려는 일을 표현 한다. 만약, 매크로가 단순히 특정 값을 대표할 경우에는 "명사"를 사용할 수도 있다. 매크로가 통상적으로 많이 사용되는 이름을 가지는 경우도 있으며, 예를 들어, 최대/최소값을 구하는 매크로는 "MAX/MIN"과 같은 것을 사용할 수 있다. 함수의 경우에는 "동사+명사"가 주로 사용된다. 특별한 경우에는(주로 임베디드 시스템에서) 자신이 속한 모듈(Module) 이름의 약자로 앞에 추가하고도 한다. 즉, 어떤 모듈을 사용하는지 이름에서 지정한다.

모듈 별로 별도의 디렉토리를 가질 수 있기에, 디렉토리의 이름을 만드는 것도 중요하다. 개발자가 디렉토리 이름만 보고도 어떤 코드가 그곳에 있을지 예상할 수 있어야 한다. 물론, 해당 모듈이 관리되고 있는 디렉토리에는 모듈에 속하는 파일들만 있어야 한다. 따라서, 상관 없는 파일들은 다른 관련된 디렉토리로 이동시켜야 할 것이다. 사실 이 부분에서도 많은 오류가 발생한다. 즉, 개발자들은 역할과 책임에 따라 파일을 잘 분리하지 않고 코드를 짠 사람이 관리하는 디렉토리에 그냥 두려는 경향이 있다. 하지만, 이는 나중에 코드의 구조적인 문제로 이어질 수 있다.

swap_two_value.c, swap_two_value.h

마찬가지로 파일의 내부도 관리되어야 한다. 파일이 적절한 이름을 가지고 있다면, 파일의 내부에 있는 함수나 자료구조는 그 이름에 해당하는 일을 처리하기 위해서만 존재해야 한다. 관련되지 않은 이름을 내부에 가지고 있다면, 이는 역할과 책임이 모호한 상태라는 뜻이다. 따라서, 이 때는 함수나 자료구조를 따로 분리된 파일로 나누어 주어야 할 것이다. 파일의 개수가 많다고 문제가 될 것은 없다. 파일의 이름이 명확하다면, 그 역할을 충분히 짐작할 수 있으며, 어떤 함수와 자료구조가 있을지 충분히 추측할 수 있다. 사실 물리적으로 조금만 분리 하더라도 코드의 구조는 상당히 간단해 지는 경향이 있다.

함수의 파라미터들도 좋은 이름을 가져야 한다. 일반적인 변수의 선언과 마찬가지로 하려는 일에 관련된 묘사적인 이름을 가져야 한다. 함수가 무엇을 입력으로 받는지 쉽게 이해할 수 있는 이름이어야 한다. 그리고, 그 함수의 선언과 정의는 동일해야 한다. 둘이 같은 이름을 사용하지 않으면, 엉뚱한 함수를 가져다 쓸 가능성이 있기 때문이다. 비슷한 이름은 혼동을 주며 실수를 유발할 가능성이 있기에, 명확히 구분되는 이름을 사용해야 한다.

```

...
for ( int i = 0; i < MAX_RANGE; i++ ) {
    age[ i ] = weight - NORM;
}

```

...

반복문을 제어하기 위해서 사용하는 변수들은 특별한 이름을 사용할 필요가 없다. 예를 들어, "i, j, K"와 같은 변수들은 "for(), while()"문 등에서 단순히 제어를 목적으로 사용된다. 이럴 경우에는 관습적인 코딩 스타일을 따라가는 것이 좋다. 배열의 인덱스와 같은 경우에도 단순한 이름이 더 어울릴 것이다. 하지만, 만약 그 인덱스가 어떤 계산이나 자료구조에 관련된 것이 있다면, 묘사적인 이름을 사용해야 할 것이다. 이것은 개발자가 코딩 할 때 적절히 선택해야하는 사항이다.

코드("C언어"라는 표현에서 알 수 있듯이)도 일종의 언어다. 언어를 사용하는 이유는 대화를 하기 위해서다. 하지만, 그 대화의 대상을 컴퓨터로 한정하지는 않는다. 코드로 하는 대화의 대상은 다른 동료 개발자들이다. 따라서, "그, 그녀, 그것, 그들, 19600"과 같이 막연한 이름(혹은, 값)을 사용하는 것은, 시간과 공간이 떨어진 상태에서 제대로 된 대화를 나누기 어렵게 만든다. 정확한 의미를 전달하기 위해서 "의미있는 이름"을 사용해야 하며, 이름을 선택 능력이 "개발자의 역량"을 나타낸다고 볼 수도 있다. 좋은 이름은 이해하기 쉬운 코드를 만들지만, 그렇지 않은 이름은 "바보도 만들 수 있는" 컴퓨터만 이해하는 코드를 만들 뿐이다.

[모든 변수는 정의 할 때 값을 정하라.]

모든 변수는 기본적으로 값을 가지는 것이 좋다. 그 값이 일단은 잘못된 값이라고 하더라도 가지는 것이 좋다. 예를 들어, "0"을 정수값을 가지는 변수에 기본적으로 할당하는 것은 의미있는 일이다. 변수를 초기화 시키지 않고 사용하는 것은, 시스템을 불안정한 상태로 만들기에 충분하다. 따라서, 일단은 변수에 어떤 값이라도 주고, 그 값이 중요하다면 나중에 검사를 하는 것이 올바른 사용 방법일 것이다. 그렇다고 무작정 아무런 값이나 넣어주기보다는, 정수인 경우에는 "0"을, 포인터의 경우에는 "NULL"과 같은 기본적인 값들을 주는 것이 좋다.

```
int x; /* 전역 변수는 0으로 초기화가 되지만, 다른 손을 빌려야 함.*/
```

```
void function( void ) {
    int y = x;
    ...
}
```

전역 변수에 대해 값을 할당하지 않으면, 프로그램이 실행될 때 기본적으로 "0"이라는 값으로 초기화 될 것이다. 하지만, 이것은 일반적인 경우이고, 만약 임베디드 시스템과 같이 운영체제가 없고 프로그램의 로더(Loader)가 없는 상황이라면 이야기는 달라진다. 즉, "0"이라는 값으로 해당 전역변수를 자동으로 초기화 해 줄 방법이 없는 것이다. 이때는 인위적으로 부팅 할 때나 응용프로그램을 실행하기 위해서 메모리로 적재하는 과정에서 "0"으로 전역변수가 저장되는 메모리를 초기화 해주어야 한다.

```
void function( void ) {
    int x;
    int y = x; /* 변수가 초기화가 안되었다는 컴파일러 경고가 발생하거나, 컴파일이 안됨 */
    ...
}
```

지역 변수들은 스택(Stack) 영역에 할당 되기에, 프로그램이 실행 전에 변수의 위치를 파악하지 못한다. 즉, 해당하는 함수가 호출되어야 변수의 위치를 알 수 있다. 따라서, 자동적으로 그 위치를 프로그램 초기화에서 알지 못하기에 초기화도 할 수 없다. 전역 변수는 그 위치가 컴파일 시에 알려지므로, 한꺼번에 모아서 프로그램 실행 시에 "0"으로 초기화 시켜줄 수 있다.

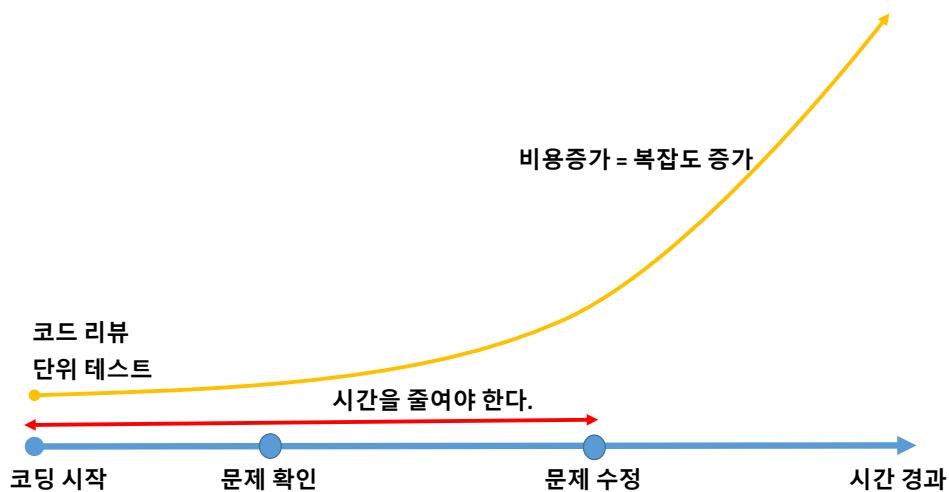
지역 변수는 사용 되기 전에 반드시 초기화 시켜주어야 한다. 만약, 그렇게 하지 않는다면, 변수가 초기화 되지 않았다는 경고가 컴파일러에서 발생하거나(“GCC”의 경우), 컴파일 자체가 안될 수도 있다(“Visual Studio”의 경우). 따라서, 지역 변수는 선언과 동시에 초기화하는 것이 좋다. 원칙적으로 변수는 사용되기 전에 반드시 초기화 되어야 한다.

전역 변수에 대한 것은 정적분석 툴에서도 검출하지 못하는 경우도 있기에, 특히 신경써서 초기화를 해주어야 할 것이다. 정적분석 툴들은 주로 컴파일러에 의지하거나, 혹은 구문 분석을 통해서 문맥(Context)을 추론한 오류를 찾아줄 수 있을 뿐이다. 따라서, 만약 검출하지 못하는 오류가 발생할 경우에는 문제를 찾아내기 어려워질 수 있다.

3. 코드 리뷰

코드 리뷰(Code Review)는 남의 코드를 읽어서 문제점을 찾아내는 활동이다. 소프트웨어 개발에서 가장 많은 버그를 사전에 발견할 수 있는 방법이지만, 그만큼의 비용도 들어가는 일이다. 코드 리뷰를 하는 방법에는 여러가지가 있지만, 일반적으로 정형적인(Formal) 코드 리뷰가 버그를 가장 많이 찾는다고 알려져 있다. 그 외에 동료리뷰(Peer Review)나 자기 책상에서 출력된 코드를 일일이 리뷰하는 방법들이 있지만, 대체로 온라인 상에서 툴을 이용해서 코드 리뷰하는 것을 많이 사용한다.

코드 리뷰를 하라는 윗 사람의 지시에 대해서 고민해 본 적이 있을까? 윗 사람이 지시는 하지만, 아래 사람은 그 지시에 대한 최선의 해답을 찾지 않고 일단은 모면할 수 있는 이유를 찾게된다. 이유는 단순하다. 코드 리뷰를 왜 하는지 이해하지 못하고, 어떻게 하는지도 알지 못하기 때문이다. 그리고, 더 중요한 것은 그런 시간이 있으면 코딩에 더 집중하는 것이 좋다고 생각한다. 하지만, 코드 리뷰도 일종의 피드백(Feedback)이라는 것을 모르고 하는 소리다. 문제는 발생한 순간과 가장 근접한 시간에 수정하는 것이 수정 비용이 적게든다. 문제가 발생한 이후에 시간이 흘러버리면, 문제의 복잡하게 변하고, 파급효과는 더 커지고, 수정비용도 시간에 비례(혹은 지수적으로 비례)해서 더 많이 들어간다.



코드 리뷰의 목적은 "코드의 가독성(Readability)" 높이고, 코드의 버그를 찾아내는 것이다. 즉, 코드가 제대로 일을 하고 있는지를 쉽게 파악하고, 구조적인 문제가 없는지를 보기 위한 것이다. 이와 같은 활동을 통해서 소프트웨어 개발자들 간에 기술의 상향 평준화를 이룰 수 있으며, 전체적인 시스템의 이해를 높여 더 높은 품질을 가지는 소프트웨어를 만들수 있다. 따라서, 코드 리뷰는 문제의 근원인 코드를 직접 보고, 코드에 대한 공통된 해석을 모든 소프트웨어 개발자들이 공유하도록 해야한다. 시간이 부족하다고 빼먹고 지나가면, 나중에 문제가 발생했을 때는 코드를 파악하는 활동과 더불어 버그를 찾는 시간까지 같이 추가적으로 소비해야 한다. 코드 리뷰는 시간을 낭비하는 일이 아니라, 시간을 효과적으로 사용할 수 있도록 해주는 일인 것이다. 코드리뷰의 주요 활동은 아래와 같다.

1. 코딩 룰(Coding Rule)을 준수하는지 확인한다.
2. 코드의 가독성(Readability)이 낮은 부분을 확인한다.
3. 코드가 주어진 일(Logic)을 충실히 수행하는지 확인한다.
4. 코드가 주어진 일 이외에 수행하는 일이 없는지(Cohesion) 확인한다.
5. 코드가 적절한 테스트(Testability)가 있는지 확인한다.

위와 같은 5가지 코드 리뷰는 체크 리스트는 소프트웨어 개발자들이 코드 리뷰 시 활용해 볼 수 있는 것들이다. 여러 개발자가 소프트웨어 개발에 참여하더라도 마치 한 사람이 코딩을 한 것처럼 보이도록 만

드는 것이 좋다. 이렇게 보이기 위해서는 팀원들이 공유하는 개발 문화를 가지는 것이 필수다. 그리고, 코드 리뷰에 앞서 팀원들이 공유하는 코딩 룰을 간단하게라도 유지할 필요가 있다. 대부분의 소프트웨어 과제는 시작 전에 이런 부분들에 대해서 잘 정리한다. 각종 문서의 양식(Format)을 정리하고, 코딩에 대한 룰도 만들어서 팀원들이 익숙하게 사용할 수 있도록 템플릿(Template)도 제공한다. 물론, 이런 것들을 만드는데 사소한 의견 충돌이 있을 수는 있으나, 한번 만들어진 규칙에 대해서는 모든 팀원이 과제 완료 시점까지 일관되게 지켜야 한다.

코드 리뷰는 일방적인 지시가 아니다. 물론 경험의 차이는 있지만, 그런 경험을 서로 공유하는 것이지 숙제를 검사받는 자리는 아니다. 코드 리뷰에서는 사람에 대한 평가가 아니라, 코드에 대한 평가를 해야한다. 이를 통해서 팀원들간의 공유되는 지식이 늘어나게 되며, “좋은 코드란 무엇인가”에 대한 의견을 나누는 것이다. 리뷰 요청을 팀내 모든 개발자들이 받는 것이 좋으며, 최소한 2명 이상이 리뷰를 완료한 후에 코드를 저장소에 "Check-In"할 수 있도록 해야 한다. 팀 전체 개발자가 5에서 9명 사이라고 한다면, 리뷰를 요청한 팀원에 대해서 2명을 고정적으로 할당할 수도 있을 것이다.

직급이나 경력은 일단 리뷰자 할당에서 무시해야 한다. 직급이 높은 사람에게 리뷰 요청이 몰리거나, 혹은 일방적인 명령하달이 되지 않도록 하는 것이 중요하기 때문이다. 현실적으로 경험을 무시할 수는 없지만, 업무가 특정한 사람에게 몰리지 않도록 만들어야 한다. 개인적으로 파트너를 정해서 하는 것보다 순환적으로 리뷰를 실시하는 것을 선호하는 편이다. 또한, 코드 리뷰가 업무 시간 중 상시 하지 않도록, 특정 시간에 리뷰 요청을하도록 하는 것이 좋다고 본다. 물론, 짹 프로그래밍(Pair Programming)과 같은 경우에는 코딩과 동시에 코드 리뷰도 일어난다고 보는 것이 좋을 것이다.

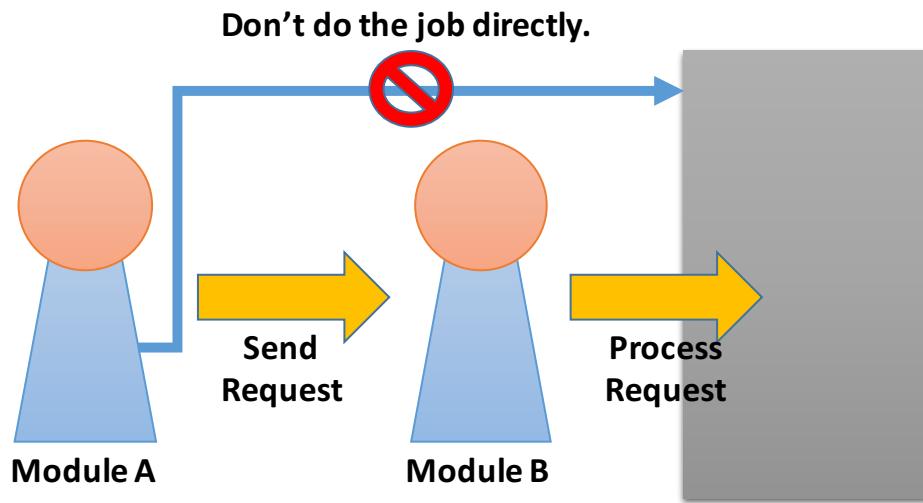
[읽기 쉬운 코드가 좋은 코드다.]

단순하지만 명확한 진실은 "읽기 쉬운 코드가 좋은 코드"라는 것이다. 즉, 코드를 작성하는 이유는 사람이 읽기 위한 것이지, 컴퓨터가 읽기를 원해서가 아니다. 프로그래머가 흔히 하는 실수가 바로 이것에 있다. 전문 프로그래머가 아닌 개발자들은 자신이 읽을 수 있는 코드를 만들기는 하지만, 남까지 고려해서 코딩 하지는 않는다. 물론, 시간이 훌러가면 자신도 "남"에 속한다는 사실을 알게 되지만, 그때가 되면 이미 늦어버린 후다. 코드는 개발 초기부터 “가독성(Readability)”을 놓쳐선 안된다.

코드가 읽기 쉽게 만들어지기 위해서는 "이름" 선택을 적절히 해주어야 한다. 즉, 상수, 변수, 함수, 파일, 디렉토리, 전체 프로그램 등등 다양한 부분의 이름을 읽기 쉽고 이해하기도 쉽게 만들어야 한다. 읽기 쉬운 코드는 이름이 의미하는 바가 명확하고, 그 이름 이외의 역할에 해당하는 일을 하지 않는다. 좋은 이름은 자신의 역할을 명확히 한정적으로 표현할 수 있어야 한다. 또한, 좋은 이름은 자신이 사용되는 범위도 지시할 수 있는 경우가 있다. 따라서, 어떤 범위를 벗어나서 사용 되지 않는다는 것도 명확히 알려 준다.

이름을 정해주었다면, 이제는 코드를 짧게 만드는 것이 가장 핵심이 되는 일이다. 여기서 말하는 "짧은 코드"란 논리적으로 한가지 일만 처리하는 코드를 만들라는 것이다. 즉, 덩어리가 큰 코드는 한번에 이해하기도 어렵거니와 수정하기도 힘들기 때문이다. 따라서, 작은 코드의 조각들로 큰 조각을 맞추는 형태의 코딩이 필요하다. 물론, 처음부터 이렇게 만들기는 어렵다. 따라서, 긴 함수를 만들었을 경우, 이를 논리적으로 한번에 한 가지 일만 충실히 수행하는 조각들로 나누어야 한다. 작은 조각들이 모여서 큰 그림을 완성하는 것과 같은 형태가 좋다.

조각을 맞추는데는 큰 그림이 필요하다. 조각 맞추는 퍼즐을 해본 경험이 있다면, 각각의 조각이 어디에 위치해야 할지를 알아야 퍼즐을 빨리 풀 맞출 수 있다. 따라서, 읽기 좋은 코드는 큰 그림을 가지고 만들어져야 한다. 큰 그림을 그리기 위해서는 시스템을 구성하는 요소들을 생각해야 한다. 이런 요소들은 그냥 만들어지는 것이 아니라, 사용자의 요구사항을 구현하기 위해서 필요한 것들로 구성된다. 즉, 사용자의 요구사항에서 필요한 구성 요소들에 대한 추출이 있어야 한다.



구성요소들 중에는 요구사항에 등장하는 것들도 있지만, 그 요구사항을 특정 시스템에 구현하기 위해서 필요한 요소들도 있다. 물론, 이런 요소들은 다른 것으로 대체될 가능성이 있는, 하나의 계층을 이룰 수 있는 것들이다. 구성 요소들을 찾았다면, 이제는 각 구성 요소에 배치할 함수를 정하는 일을 해야한다. 특정 요구사항을 구현하기 위해서 각각의 구성요소들이 어떤 역할(Role)을 해야하는지 정하고, 동작 시나리오에 맞는 함수들을 하나씩 정하도록 한다. 이때 중요한 것은 자신의 역할이나 책임 범위가 아닌 경우, 다른 구성 요소에 "요청(Request)"해야 한다는 점이다. 코딩을 하다보면 지금하고 작성하고 있는 모듈의 역할 범위를 벗어나는 부분에 대해서도 함께 작성하는 경향이 있기 때문이다.

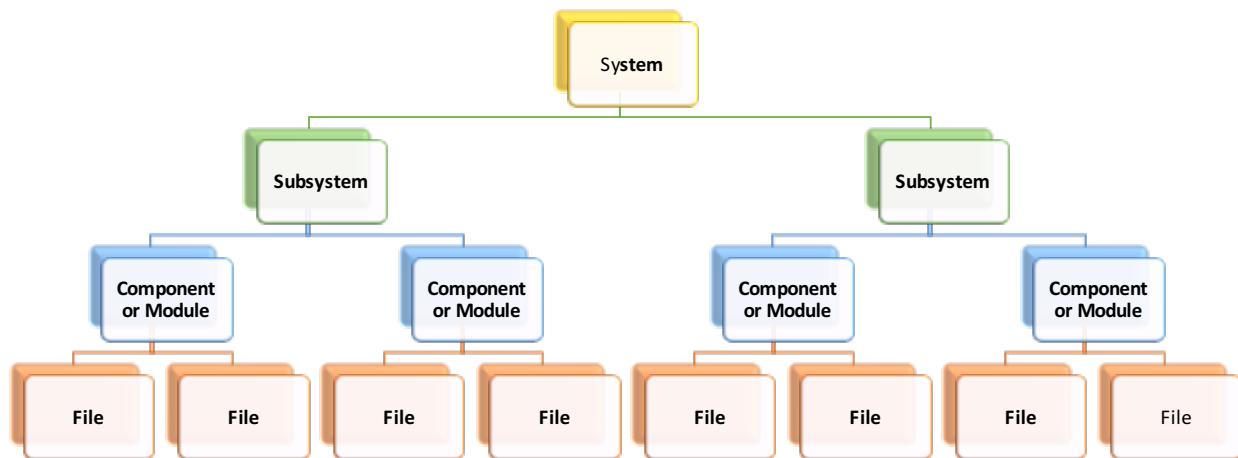
읽기 좋은 코드는 "좋은 이름을 가지는" 구성요소들의 "역할과 책임 범위가 명확" 보인다. 즉, 각각의 구성요소들이 해야할 일만 충실히 수행하는 것처럼 보인다. 해야하는 일이란 정해진 "역할과 책임(Role & Responsibility)"이라는 것으로 표현되며, 이것을 구성요소가 "자율적"으로 수행하는 것이다. 이렇게 만들어진 코드는 당연히 의존성이 최소화 된다. 다른 구성요소들이 하는 일을 애써 처리할 필요가 없기에, 서로가 서로에게 제공하는 인터페이스를 주요 대화 창구로 활용할 것이다.

어떻게 구현할 것인가를 정하는 것은 가장 마지막에 해야할 일이다. 즉, 구현 전에 구성요소간의 관계를 만들어주고, 그 관계를 유지하는 것을 전제로 요구사항을 구현하는 시나리오를 만든다. 시나리오를 구현할 수 있는 인터페이스 역할을 하는 함수를 정의하고, 이렇게 정해진 함수들을 구현하는 것이다. 구현하면서 구조화 시키는 것은 어렵다. 하지만, 구조화 시킨 후에 구현하는 것은 쉽다. 구현이 완료되는 조건은 테스트가 완료되는 시점과 같다고 보면, 테스트를 먼저 만드는 것도 좋은 선택이다(TDD : Test Driven Development 같은 방법론을 활용하는 것).

구성 요소를 찾는 것이 어렵다고 생각되면, 주어진 문제를 분석하는 과정을 반복해야 한다. 이때, 사용하는 용어를 사용자의 것으로 하는 것이 중요하다. 개발자의 용어는 개발자의 관점만을 반영하기기에, 사용자의 관점과는 차이가 있을 수 있다. 또한, 문제를 이해하고 같이 대화를 나누기 위해서도 사용자의 용어를 사용하는 것이 좋다. 그 용어 속에서 시스템을 구성할 요소들을 찾아내고, 그들간의 관계를 맺어주는 것이다. 기본적인 관계가 만들어지면, 나머지는 시나리오를 구축하는데 필요한 각자의 역할이 무엇인지 정의하는 것이다. 물론, 구현 측면에서 실제로 동작하게 될 환경에 대해서도 구성요소를 발견할 수 있을 것이다.

역할은 반드시 정의해야 하며, 정의된 역할 범위를 넘어서는 일을 처리하면 안된다. 또한, 그 역할에 적절한 이름을 갖추는 것도 반드시 필요하다. 만약, 자신의 역할 범위를 넘어선다면, 다른 구성 요소(모듈이라고 하겠다)에게 "요청"을 전달하는 것이 필요하다. 모든 것을 혼자서 다 하기보다는 역할과 책임이 나누어진 모듈을 이용해서 분해하고, 다시 결합하는 과정으로 해법을 찾는 것이 소프트웨어를 제대로 만

드는 방법이다(소프트웨어의 복잡함을 적절한 수준에서 관리하는 방법). 나누어지지 않은 역할과 책임은 결국 소프트웨어의 내재된 복잡함을 더할 뿐이다.



코드의 배치도 중요하다. 코드는 각각의 디렉토리, 파일 들에 적절하게 분포되어 있는 것이 좋다. 너무 하나의 디렉토리나 파일에 치중되어 있으면, 복잡한 역할을 하는 모듈이나 파일이 존재한다는 뜻이 된다. 따라서, 마치 나무의 뿌리가 골고루 땅속으로 퍼져나가듯, 각각의 모듈과 파일들에 나누어져 존재하도록 만들어야 할 것이다. 물론, 부분적으로 복잡하게 구현되어야 할 모듈도 있을 것이다. 하지만, 그런 모듈도 상위 수준에서 봤을 때는, 고르게 분포된 코드를 가질 수 있도록 만들어 주는 것이 코드를 이해하는데 도움이 될 것이다.

[책임(혹은, 계약)과 구현의 분리]

책임은 "무엇을 할 수 있다는 개념을 말한다. 즉, 그러한 것을 맡을 수 있다는 것이다. 따라서, 그 책임을 가지고 있는 모듈은 그 일을 하는 것이 책임을 완수하는 것이다. 그것을 어떻게 완수할 지에 대해서는 전적으로 그 일을 실행하는 모듈의 책임하에 있다. 따라서, 일을 지시하는 측면에서는 누구에게 일을 맡겨야 할지를 결정하는 것이 중요하지, 어떻게 그 일을 처리해야 하는지 알 필요가 없다. 일을 어떻게 처리하는 것이 좋은지는 그 일을 처리할 책임을 맡은 모듈에서 결정할 문제이며, "어떻게"에 대해서 간섭받으면, 모듈은 내부적인 구현을 외부로 노출할 수 밖에 없다.

상세한 것(구체적인 것)에 의존하면 할수록 변경에 대해서 취약해 진다. 하지만, 대부분의 프로그래머는 이런 것에 대해서는 별로 신경쓰지 않는다. 필요하다면 만드는 것에 익숙하기 때문이다. 즉, 특정 모듈이 해야 할 일을 자신이 필요하다는 이유로 직접 구현한다. 물론, 이렇게 만들어진 코드의 대부분은 역할과 책임이 범위가 모호하게 되는 경우가 많다. 결과적으로 특정 모듈에 있어야 할 기능들이 다른 모듈의 일부로 구현된 경우를 흔히 보게 된다.

이런 현상이 발생하는 이유는 앞에서 이야기 했듯이 "역할과 책임"이 모호하기 때문이다. 그리고, 지나치게 세부적인 구현에 의존적인 코드를 만들기 때문이다. "성능상의 이유"라는 말로 내부의 데이터 구조를 직접 접근하고 싶은 욕구를 극복하지 못하기 때문이다. "직접 해야 할 일과 요청해야 할 일"을 구분할 수 있어야 한다. 즉, 설계상의 정의된 역할과 기능만 충실히 수행해야 하지만, 구현의 편의로 한 사람이 맡은 모듈의 역할이 여러가지가 되는 경우는 너무나 흔하다.

이를 극복할 수 있는 방법은 "명확한 역할과 책임"을 정의하는 것이다. 그리고, 그렇게 정의된 역할과 책임을 만족할 수 있도록, 각 모듈의 "인터페이스"를 책임과 역할에 맞게 정의하는 것이다. C언어의 인터페이스는 함수다. 따라서, 각 모듈이 외부에 공개할 함수를 정의해서, 자신의 책임 범위를 명확히 알려주는 것이다. 이때, 세부적인 데이터를 숨기는 것은 필수다. 내부에서 사용하는 자료구조의 변경이 외부로 영향을 주지 않아야 한다. 당연히 직접적으로 자료구조에 접근하는 것을 허용해선 안된다.

일반적으로 복잡한 문제를 복잡하게 푸는 것은 대부분의 개발자들이 다 할 수 있는 수준의 문제 풀이다. 복잡한 문제를 역할과 책임이 명확한 모듈로 분해하고, 이를 하나씩 해결하는 것이 좋은 개발자가 해야 하는 진정한 일이다. 인터페이스는 역할과 책임의 범위를 규정하기에, 적절한 수준의 해야할 일(역할)을 정의하는 것이 좋다. 좋은 인터페이스는 한번에 한가지 일만 충실히 수행하는 것이다. 따라서, C언어에서의 함수는 한 번에 한 가지 정의된 일만 잘 해주도록 만들어야 한다. 여러 역할을 책임지는 인터페이스는 제대로 동작하지 않을 가능성이 높다(버그가 발생할 가능성이 높다).

함수는 내부와 외부로 나누어진다. 주로, 인터페이스 측면에서 중요한 것은 외부에 제공되는 함수들이다. 이런 함수들은 상세한 내부 구현을 감추어야 하기에 추상적인 이름을 가지는 것이 좋다. 각각의 함수가 특정 순서를 지켜서 호출되어야 한다면, 내부적인 자료구조(상태)도 이에 관련되었을 가능성이 높다. 따라서, 각각의 함수는 될 수 있으면 독립적으로 동작할 수 있도록 만드는 것이 좋다. "일을 어떻게 하는지 지시하려고 하지말고 요청하라"고 하는 이유가 여기에 있다. 상세한 구현을 알려고하지 않고 해당 역할을 하는 모듈에 책임을 전적으로 맡겨야 한다. 그것이 모듈과 모듈의 관계를 좀 더 유연하게 만들어줄 것이다.

```
unsigned int getPrice( unsigned int productID );
unsigned int getVolume( unsigned int productID );
```

위와 같은 두개의 함수를 제공하는 모듈이 있다고 가정할 때, 각각은 물건의 총 값을 계산하기 위해서 호출되어야 한다. 즉, 물건 값을 계산하기 위해서, "getPrice(productID) * getVolume(productID)"와 같이 호출될 것이다. 하지만, 이것은 내부적인 정보를 외부에 드러내는 행동이다.

```
unsigned int getTotalPrice( unsigned int productID );
```

위와 같이 함수를 사용한다면, 내부에서 관리되는 정보에 상관없이 "원하는 일을 요청"하는 형식으로 인터페이스를 정의할 수 있다. 또한, 추가적으로 어떤 "의도"를 가지고 있는지도 확실하게 모듈에 전달할 수 있다. 호출받은 모듈은 자신이 관리하는 내부 정보를 이용해서 요청한 일을 처리한 후, 결과만을 돌려줄 것이다. 물론, 이때는 내부 정보 자체를 관리하는 역할이 해당 모듈에 있게되며, 정보를 초기화하는 일과 요청된 일을 처리하는 역할을 함께 가질 것이다.

이것을 일반화 시키면, 자료구조가 정의된 곳에 자료구조를 조작하고 연산하는 함수들이 같이 정의되어야 한다. 그래야만 자료구조에 대한 직접적인 조작과 연산이 발생하지 않는다. 또한, 자료구조 자체가 내부로 감춰질 수 있다. 물론, 이런 것들은 이미 객체지향 언어에서는 일반화된 방법이다. C언어가 객체지향을 지원하는 언어는 아니지만, 개념적으로 필요한 부분들을 이용해서 응집성이 높고 의존성이 낮은 코드를 만드는 것은 가능하다. 결국 코드란 복잡함을 외부에 드러내지 않고 내부에서 갈무리 할 수 있도록 해야한다.

[코드 리뷰를 못하는 이유?]

코드 리뷰를 제대로 못하는 것은 "개인의 의지"와 관련이 있다. 사실 코드 리뷰를 의무화하기 위해서는 어느 정도 강압적인 분위기를 이끌어야 할 필요도 있지만, 결국 "재대로" 하기 위해서는 코드를 작성한 사람들이 "의지"를 가져야 가능하다. 시스템을 구축하고, 리뷰 된 코드의 범위를 확인하는 각종 도구를 동원 하더라도, 내부적으로 개발자들이 제대로 하는지 알기는 쉽지 않다.

개발자들은 각종 도구의 체크를 교묘하게 피해나가는 방법을 쉽게 찾을 수 있으며, 바쁘다고 생각되면 그런 것들을 너무나도 쉽게 이용한다. 예를 들어, 테스트 케이스를 늘리라고 하면, 아마 일주일도 못가서 두 배로 늘었다고 자랑할지도 모른다. 물론, 제대로 된 테스트 케이스가 늘어나지는 않았을 것이다. 단지, 테스트로 입력되는 값들을 의미없이 조작 했을 수도 있다. 따라서, 코드 리뷰를 실시하는 개인들이 공통된 의견과 의지를 갖추고 있지 않다면, 단순한 구호나 보여주기 정도 밖에 되지 않을 것이다.

코드 리뷰는 시간이 필요하다. 이런 시간들이 전부 개발하는 시간에 쏟아부으면 더 빨리 개발할 수 있다고 믿는다면, 잘못된 생각을 가진 것이다. 코드 리뷰는 개발의 일부이지, 시간이 남을 때 하는 여가 활동이 아니다. 따라서, 코딩과 같이 일로 취급 해야지, 취미처럼 해도 되고 안 해도 상관없는 일이 아니다. 단위 테스트와 마찬가지로 코드의 품질을 높일 수 있는 방법들은 전부 일이다. 일이 아니라고 생각하기 때문에 시간이 없다는 이유로 생략되고 마는 것이다.

“우물에서 승능”을 찾을 수 없듯이, 제대로 검토되지 않는 코드를 전체 코드에 통합하는 것은 오히려 시간을 지연시킬 뿐이다. 지연되는 시간을 조금이라도 줄이기 위해선, 사전에 버그를 제거하는 것이 가장 효과적인 방법이다. 버그는 발생하는 시점과 그것을 찾는 시점이 빠를수록 추가적인 비용이 줄어드는 경향이 있다. 그리고, 늦게 발견되면 발견될 수록 그 비용은 선형적인 것이 아니라, 기하급수적으로 증가한다. 이른 시간에 찾을 수 있는 방법을 안쓸 이유가 없는 것이다.

어떻게 할는지를 몰라서 코드 리뷰를 하지 않는 경우도 있다. 무엇을 리뷰에서 확인해야 하는지 규정이 없다는 것이다(물론, 규정은 항상 최소한만 정의할 뿐이지만). 리뷰하는 사람 입장이라면 자신이 리뷰하고 있는 코드의 문제점을 찾아서 이를 원작자에게 알려야 한다. 하지만, 어떤 것을 주의 깊게 볼지 미리 합의하고 시작하는 것이 좋다. 그런 것들이라면 리뷰하는 사람도, 그것을 받는 사람도 부담없이 생각할 수 있기 때문이다.

리뷰에 들어가기 위한 조건으로는 정적분석 이상 유무, 코딩률 준수 여부, 복잡도 분석이나, 함수 LOC(Line of Code), 복제된 코드의 존재 여부 등의 기준이 맞는지 확인할 수도 있다. 단위 테스트의 존재 여부 및 커버리지도 필요할 수 있다. 리뷰 시에는 그런 것들은 기계적으로 다 검증되었다고 생각하고, 논리적인 오류나 가독성을 위주로 진행하면 될 것이다. 도구가 필요하면 "Review Board"나 "Code Collaborator"와 같은 툴을 사용할수도 있을 것이다. 툴은 도구일 뿐이며, 리뷰는 사람의 정성적인 결과물이다. 따라서, 무엇을 할지 정하면 객관적인 기준으로 코드에 대한 비평을 할 수 있어야 한다.

코드 리뷰를 정기적으로 하겠다는 말은 코드 리뷰가 상시적으로 발생하는 것이 아니며, 특정 이벤트로 그칠 가능성이 높다. 차라리, 코드를 작성하는 시점에 동료가 리뷰(Peer Review)하는 것이 좀 더 리뷰를 빠르고 격식이 없이 진행되도록 만들어 줄 것이다. 정기적인 리뷰는 중요한 몇 개의 모듈에 대해서 모든 팀원이 알아야 할 것으로 한정하면 좋을 것이다. 독립적으로 다른 코드에 큰 영향을 주지 않는다면, 전체적인 인력이 참여해서 리뷰 할 필요가 없을 것이다. 따라서, 리뷰 될 코드의 종류에 맞춰서 공식으로 할지, 아니면 비공식적으로 할지를 정하면 된다.

코드 리뷰를 할 때는 투자되는 시간에 비해 좋은 성과(많은 버그를 찾아내는)를 만들어 내기 위해서, 시간당 500라인 이하로 리뷰하는 것이 좋다. 너무 오랜 시간을 하는 것은 바람직하지 않으며, 들어가는 비용에 비해서 찾아내는 버그의 수도 증가하지 않는다. 따라서, 500라인 정도의 코드가 불어날 수 있는 시간을 주기로 해서 한 시간정도 사람들이 모여서 코드를 공식적으로 리뷰하는 것이 좋다. 또한, 한 번의 코드 리뷰를 진행할 때 2시간 이상은 하지 않는 것이 좋다.

코드 리뷰 이후의 활동에 대해서 알지 못하면, 리뷰의 결과물이 제대로 반영되지 않을 가능성이 높다. 리뷰의 결과물은 코드에 반영되어야 하며, 나중에 다시 정확히 반영되었는지 리뷰를 통해서 확인해야 한다. 나중에 원인 분석을 위해서 로그(Log)를 남길 때, 특정 포맷(Format)을 사용하는 것이 좋다. 버그에 대한 개선이나 가독성, 혹은 더 좋은 코드나 기능추가, 리팩토링의 결과물인지 표현할 수 있으면 된다.

코드의 미진한 점은 보완하도록 요구하고, 리뷰를 마치지 못한 코드라도 개발의 편의상 개발상의 코드와 통합될 수도 있다. 만약, 여러 단계로 코드를 관리하는 시스템을 가지고 있다면, 배포 코드와 개발 코드를 분리해서 유지하고, 개인 코드도 나누는 것이 좋을 것이다(Multi-Stream). 100%의 코드를 리뷰하는 것도 좋지만, 중요한 코드들을 집중적으로 리뷰하는 것도 가능하다. 즉, 복잡도가 높고 변경이 잦은 코드가 있다면, 그런 것들을 중심으로 리뷰를 강화할 수 있다.

코드 리뷰는 시간이 부족하다고 생략할 수 있는 활동이 아니며, 버그를 사전에 방지할 수 있는 가장 효과적인 방법이다. 단지 그것을 실시하는 방법의 차이가 있을 뿐, 반드시 해야 할 "개발 활동"이라는 점에서는 모두 동의를 한다. 물론, 그것이 중요하지 않다고 생각하는 일부 관리자들의 눈에는 코드 리뷰가 못마땅하게 생각될 수도 있다.

해야 할 일을 제때 하지 않는 것은 비용의 추가 발생을 가져오게 되며, 조금쯤 빛을 갚아나가는 마음(실제로는 빛 보다는 이익이라고 봐야겠지만)으로 적극적으로 코드 리뷰를 해야 한다. 코드 리뷰는 버그를 사전에 차단하기에 과제 일정에 도움을 줄 수 있으며, 들어가는 비용에 비해서 더 큰 혜택을 준다. 물론, 그렇다고 모든 버그를 없애지는 못하지만, 찾아낼 수 있는(줄일 수 있는) 방법들은 다 사용해야 할 것이다.

[코드 리뷰 하는 방법]

코드 리뷰는 오류를 가장 많이 찾아낼 수 있는 방법이다. 하지만, 대부분의 개발자들이 가장 어려워하는 것이기도 하다. 테스트 단계에서 찾아내는 버그와 코드 리뷰에서 찾아내는 버그의 수를 비교한다면, 생산성이 높은 조직은 코드 리뷰에서 찾아내는 비율이 더 높을 것이다. 물론, 전체 버그(혹은, 문제점 [defect]을 관리한다면)를 시스템으로 등록하고 그것을 추적할 수 있다면 그럴 것이다. 대부분의 경우 코드 리뷰에서 찾아내는 문제점은 따로 보고되지 않는다. 테스트에서 발생하는 버그는 주기적으로 보고되지만, 코드 리뷰에서 수정되는 것들은 팀 내부에서만 알고 있다. 이런 상황에서는 보고 받는 쪽의 관점에서는 코드 리뷰라는 활동의 중요성이 떨어지게 된다. 따라서, 소프트웨어 개발 주기상의 모든 "오류 (Defect)"이 관리되고 있다는 것을 보여줄 필요가 있다. 물론, 이런 것들은 자동화를 동반하기에 비용이 들어갈 수도 있지만, Bugzilla나 Mantis, Trac 등을 이용해서 무료로 할 수도 있다. 필요하다고 생각되면 방법은 충분히 찾을 수 있다.

코드 리뷰의 일반적인 프로세스는 "저장소에서 코드 가져오기->코드 수정하기->다시 원본과 통합해서 빌드 및 테스트하기->동작할 경우에 코드 리뷰 요청하기->원본 코드와 수정된 코드 비교해서 검토하기->문제가 없을 경우 저장소에 코드를 넣는 것을 동의, 혹은 문제가 있을 경우 수정 요청->문제가 해결될 때까지 반복->코드를 저장소에 넣기"와 같다. 즉, 코드를 원본 저장소에 통합하기 전에 리뷰를 하는 것이다 (Pre-commit Review). 이런 과정으로 해야하는 것이 정석이지만, 대부분의 경우 코드를 저장소의 원본에 통합한 다음 리뷰를 하는 것(Post-commit Review)을 선호한다. 개발자들은 전자의 경우 개발이 늦어진다는 불평을 하기 때문이다. 물론, 후자의 경우 원본 코드가 깨져서 동작하지 않을 위험성이 있으며, 그럴 경우 통합 및 테스트, 코드 수정의 재작업이 발생할 가능성이 더 높다. 빨리 할 수 있다고 생각할 수도 있지만, 그렇다고 비용이 전혀 없는 것은 아니기 때문이다. 뭔가 얻는 것이 있으면, 그것으로 인해서 치루어야 할 비용도 있다.

보통 하기 힘든 일은 잘게 나누어서 자주하는 것이 해결 방법이다. 코드 리뷰도 마찬가지다. 한번에 많이 하려면 힘들다. 통계적으로 정리한 결과에 따르면 한번에 400라인 이하의 코드를 리뷰하는 것이 좋고, 시간을 길게하는 것보다 60분 내로하는 것이 좋다고 한다(공식적인 코드리뷰에서). 그렇게 하기 위해서는 온라인 리뷰와 오프라인 리뷰를 병행해야 하며, 온라인 리뷰는 상시 변경과 같은 부분에 집중하고, 오프라인 리뷰는 중요한 변경이나 신규 기능 추가와 관련된 부분에 집중하는 것이 좋다.

한번에 많은 코드를 수정하고 저장하는 것보다 작은 변경이라도 자주 리뷰하는 것이 효과적이다. 많은 코드를 같이 리뷰하는 것은 생각보다 어렵고 준비해야 할 것도 많기 때문에, 개발자들이 낭비라고 생각할 가능성이 높고 집중력도 저하된다. 가장 좋은 것은 코딩과 병행하는 리뷰다. 즉, 짹 프로그래밍(Pair Programming)과 같은 XP에서의 실천 방법을 가져오는 것이다. 버그가 발생하는 원류 단계에서 버그를 제거하는 것이 가장 비용이 싸기 때문이다. 아예 버그를 미연에 방지한다면 좋겠지만, 코딩은 창의성이 포함된 부분이라 그런 것이 불가능하다. 물론, 즉각적인 피드백은 가능한 부분이기에 빨리 버그를 찾는 것이 중요하다.

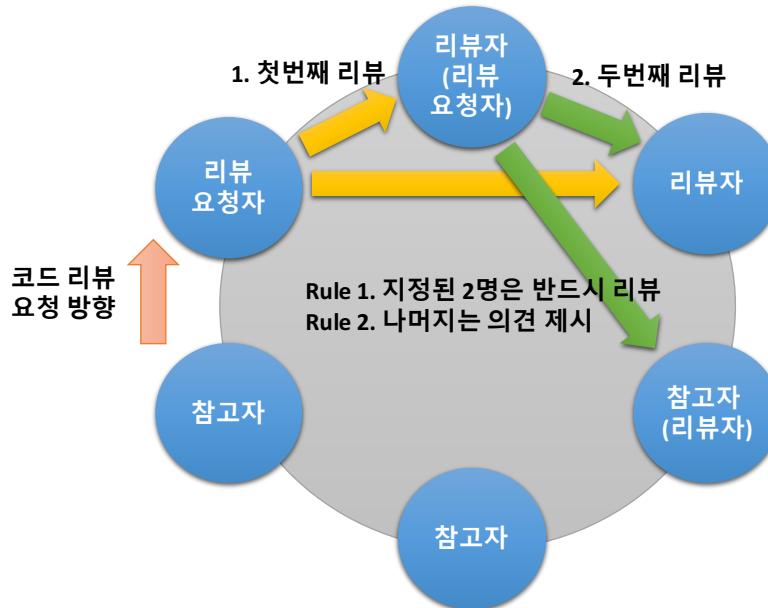
코드 리뷰는 사람과 사람사이의 일이다. 즉, 어떤 사람이 만든 결과물을 다른 사람이 검토하는 것이다. 그리고, 그 반대 역할을 번갈아 가면서 맡을 수 있다. 이때 생기는 문제점은 코드와 자신을 동일시해서

코드의 문제점을 이야기하는 것을 자신을 비난한다고 생각할 수 있다는 것이다. 따라서, 이런 것들을 방지하기 위해서는 객관적인 기준을 만들어서 적용할 필요가 있다. 코딩 스타일과 같이 들어 쓰기나 팔호의 사용과 같은 것들은 편집에서 사용하는 툴에서 제공하는 기본 포맷을 활용하는 것이 좋다. 이런 것으로 왈가왈부하는 것은 옳지 않다.

대신에 코딩 룰에서 정하고 있는 문제를 사전에 예방할 수 있는 기법이나, 변수나 함수등에 대한 이름을 주는 규칙에 대해서는 리뷰에서 보는 것이 좋다. 읽기가 어색한 코드나 잘 이해가 안되는 코드에 대해서는 수정하도록 요청할 수 있다. 이런 내용들을 점검 리스트처럼 만들어서 가지고 있다면, 좀더 객관적으로 코드를 검토할 수 있고, 검토 당하는 입장에서도 편하게 받아들일 수 있을 것이다. 무작정 어디가 잘못된 것 같다고 지적하기보다는 합의에 의해서 도출된 체크 항목(Checklist)을 가지고 이야기하는 것이 편할 것이다.

경험이 많은 사람만 코드를 리뷰하는 것은 옳지 않다. 경험이 있는 사람도 자신이 작성한 코드를 다른 사람이 리뷰 할 수 있도록 해야한다. 어느 누군가에게만 집중적으로 리뷰를 받아야 한다면, 리뷰하는 사람의 업무가 과다해 질 가능성이 있다. 또한, 한 사람에게만 리뷰를 받았다고 코드를 저장소에 넣는 것도 문제가 있다. 상시적인 작은 변경에 대해서는 그렇게 할 수 있다고 하더라도, 중요 기능의 변경이나 새로운 기능의 추가와 같은 것들은 여러 사람이 리뷰를 봄해야 할 것이다. 그리고, 최소한 2명 이상은 동의를 해야만 저장소에 넣을 수 있도록 해야한다.

누군가가 출장이나 결근 했다고 리뷰가 돌아가지 않아서도 안되기에, 항상 리뷰를 하는 사람들의 순서는 정하고 있는 것이 좋을 것이다. 만약, 누군가가 지금 회사에 없다면 다른 사람이라도 리뷰를 할 수 있도록 해야할 것이다. 기능 전체를 다 만들었다고 리뷰를 하는 것도 좋지 않다. 너무 작은 단위의 변경 까지도 전부 리뷰하는 것도 좋지 않다. 즉, 적절한 도막을 낼 필요가 있다는 뜻이다. 상시 리뷰라면 20분 내외에 끝낼 수 있는 분량으로 하는 것이 좋을 것이다. 중요 기능의 변경이나 추가는 오프라인 상에서 400라인 60분을 넘어선 안된다.



리뷰는 일이다. 리뷰를 성가시거나 부가적으로 해야할 숙제로 생각하는 것은 곤란하다. 리뷰는 개발과정이기에 당연히 일정에 반영되어야 한다. 설계, 코딩, 테스트, 리뷰는 하나의 묶어진 과정이라고 봐야 한다. 각각을 따로 분리해서 생각하면 생략이 발생하기 마련이다. 이런 것이 반복되면 결과적으로 코드 리뷰는 형식으로 그칠 가능성이 높다. 이를 측정하기 위해서는 평균적으로 리뷰하는 시간이 얼마나 되는지를 보면 쉽게 알 수 있다. 또한, 코드의 코멘트가 달리는 비율을 보고도 판단할 수 있을 것이다. 리뷰에서 나오는 문제점(Defect)도 나중에 모아서 수치적으로 비교해 볼 필요가 있다. 측정하지 않으면 개선되지

않기 때문이다. 물론, 이런 것들을 피해갈 다양한 방법을 개발자들은 만들 것이다. 따라서, 측정된 결과를 가지고 고과에 반영하는 것은 좋지 않다. 차라리 코드 리뷰의 결과를 통해서 코드에 대한 책임을 묻는 것이 올바른 선택이다. 즉, 리뷰한 코드에서 발생한 오류에 대해서는 코딩한 사람과 리뷰한 사람 모두에게 책임을 묻는 것이다. 물론, 그렇기 위해서는 리뷰와 테스트 코드가 함께 했다는 것이 보장되어야 한다. 리뷰는 리뷰일 뿐이다. 실행해 봄아 정말 제대로 동작하는지 알 수 있기 때문이다. 따라서, 테스트 자동화도 후속으로 추진되어야 할 것이다.

리뷰를 하는 것은 프로그래머의 기본이다. 그리고, 그 기본기 중에서도 가장 중요한 핵심 사항이다. 코드 리뷰는 남을 돋는 것과 동시에 자신을 돋는 행동이다. 리뷰를 통해서 얻는 것은 남을 돋는데만 사용 되는 것은 아니다. 다른 사람의 코드를 보고 자신의 코드에 대한 지식을 넓힐 수 있기 때문이다. 리뷰가 어려운 이유는 "긍정적인 장기효과(댄 애리얼리의 "경제 심리학"이라는 책에 나오는)"보다는 "부정적인 단기 효과"에 사람들이 이끌리는 것과 같다. 즉, 장기적으로는 좋다는 것을 알면서도, 단기적으로 편한 것을 찾기 때문이다.

프로그램 개발은 장기 레이스와 비슷한 면이 많다. 빨리 뛰면 지쳐서 더 느려진다. 자기 페이스를 찾아서 꾸준히 나아갈 경우에만 결승선에 조금이라도 빨리 도달할 수 있다. 이때 리듬을 찾는데 필요한 한가지 방법이 바로 "코드 리뷰"인 것이다. 즉, 문제점을 사전에 차단해서 시간이 흘러 갈수록 쌓이는 버그로 인한 속도 저하를 없애기 때문이다. 시간이 흐를수록 속도감이 저하되고 있다고 느낀다면, 크게 두 가지 요인이 작용하고 있음을 알 것이다. 즉, 코드 검증으로 인한 속도 저하(버그 발생으로 인해서 재작업이 많아짐)와 코드내의 복잡도로 인한 코드 수정과 신규기능 추가 속도의 감소이다. 이를 방지하기 위해선 역시 코드 리뷰에서 복잡함과 문제점을 다루는 과정이 필요한 것이다.

문제를 조기에 발견하는 역량은 중요하다. 소프트웨어는 시간이 흐를수록 점점 더 하드(Hard)해져 바꾸기 어려워진다. 소프트웨어를 변경이 가능하다고 생각하는 사람들은 그 비용을 제대로 파악하지 못하고 있는 경우가 많다. 소프트웨어는 "소프트하지만 변경하는 비용은 크다"라는 것을 절대 잊어선 안된다. 재작업이 최소화가 되기 위해선 이른(Early) 검증이 필요하며, "코드 리뷰"는 이른 검증에 필요한 중요한 방법이다. 단위 테스트와 코드 리뷰만 가지고도 대략 최소 50%이상의 버그를 방지할 수 있다(물론, 제대로 한다면 그렇다는 것이다.). 그렇다면, 코드 리뷰를 하지 않는 이유인 "부정적인 단기효과"를 극복할 수 있는 방법은 무엇일까? "강제화"는 가장 마지막 수단이 되어야 할 것이다.

중요한 것은 역시 "개발자의 마인드"다. 그리고, 더 중요한 것은 과제를 리드하는 권한을 가진 사람의 품질에 대한 개념이다. 아무리 개발자가 코드 리뷰가 필요하다고 이야기를 해도, 들어주지 않는 사람이 관리자라면 소용없는 일이다. 그리고, 관리자가 아무리 코드 리뷰를 외치더라도 형식적으로 하는 리뷰로는 만족되지 못한다. 사람이 하는 일은 사람이 제대로 해야만 이루어진다. 우린 지금까지 제대로 하는 것을 "제대로" 본적이 없다. 물론, 그 이유는 자신에게 묻는 것이 가장 빠를 것이다.

[코드 리뷰 체크 리스트(Checklist)]

코드 리뷰는 단위 테스트와 함께 코드에서 버그를 사전에 막아주는 가장 좋은 방법이다. 코드 리뷰가 어려운 이유는 다양하게 있지만, 대부분의 개발자들은 "시간이 없다"는 이유로 외면하고 있는 것이 사실이다. 하지만, 시간이 없다는 말로 버그가 발생했을 때 들어가는 노력과 낭비되는 비용에 대한 정당한 이유는 되지 않는다. "제대로 할 시간은 없지만, 다시 할 기회는 주어진다."라는 말은 이런 것을 빗대서 이야기 하고 있다. 실제로 시간이 없어서 못하는 것이 아니라, 시간이 있어도 제대로 못하거나, 이기적인 이유로 안하는 경우가 더 많다.

코드 리뷰를 하기 어려운 이유 중에 하나가 객관적인 리뷰를 하기 힘들다는 점이다. 즉, 코드와 코드를 작성한 사람을 동일시해서, 코드에 비판하는 것을 작성자를 비판하는 것으로 오해하는 경우가 많다는 것이다. 이렇게 하다보면 결국 서로간의 감정싸움으로 번지게 되고, 코드 리뷰를 안하는 것이 오히려 팀워크에 도움이 될지도 모른다는 생각을 가지게된다. 그리고, 상급자가 작성한 코드를 하급자가 보는 것도

객관적인 리뷰를 못하게 만드는 요인이므로, 이런 부분도 해결해 주어야 제대로 된 코드 리뷰를 할 수 있다.

이와 같은 상황에서 적절한 방법은 먼저 코드 리뷰 시간을 줄이는 툴(Tool)을 사용하는 것과, 객관적인 점검 항목들을 만들어 코드 리뷰가 가지는 감정적인 문제들을 피해가는 것이다. 온라인 툴은 코드를 리뷰하는 사람 입장에서 모이지 않고 진행할 수 있어서 자신이 여유있을 때 코드 리뷰를 할 수 있도록 만들며, 객관화된 점검 항목들은 동의된 일의 기준에 비판이 비난이 되지 않도록 만들어 줄 것이다. 물론, 여유가 있는 시간이라는 것을 만들기는 힘들다. 자신의 코드를 작성하는 것도 힘든데, 남의 코드까지 본다는 것은 여러가지로 스트레스를 유발한다. 하지만, 만들어지는 코드는 전체의 책임이라는 관점에서 본다면, 이것도 극복 가능한 문제라고 생각한다.

코드 리뷰는 잘못된 부분을 찾고, 그것을 수정하도록 요청하는 과정으로 구성된다. 나중에 그렇게 수정된 코드는 다시 리뷰를 통과해야 하고, 지적된 사항에 대해서 수정되었는지 확인하는 절차도 거친다. 자주 발생하는 잘못된 코딩 습관과 같은 것들은 점검 리스트에 추가될 수 있으며, 에러를 유발하기 쉬운 코딩도 점검 리스트에서 관리할 수 있다. 만약, 일정 수준 더 이상 그런 오류들이 나타나지 않거나, 자동화된 툴로 점검이 가능한 부분이 있다면, 점검 항목에서 빠질 수도 있을 것이다. 사람이 눈으로 볼 수 있는 것에는 한계가 있는데, 코드 리뷰도 모든 버그를 다 찾을 수 없다. 다양한 방법을 같이 사용할 수 있다면, 그것이 최선의 결과를 좀 더 빨리 가져올 것이다.

한번에 많은 코드를 리뷰하려고 해선 안된다. 대략 1시간에 400라인 이상은 코드 리뷰를 해봐야 별로 생산성이 없다는 것을 증명한 자료도 있다. 따라서, 많은 양을 한번에 리뷰하기보다는 적은 양이라도 조금씩 자주 하는 편이 좋다. 코드 리뷰 시간도 줄일 뿐 아니라, 좀 더 코드의 변화를 작게 유지해서 빨리 버그를 찾을 수 있도록 해주기 때문이다. 코드 리뷰에서 중점적으로 보아야 할 부분은 "코드의 가독성"이다. 잘 읽을 수 있는 코드를 짜야지 버그를 찾아내기도 쉽다. 따라서, 코딩 룰이나 스타일 등은 일관성이 있어야 하며, 변수, 함수, 풀더 등의 이름도 의미를 충분히 가지도록 정의되어야 한다. 한가지 추가할 것은 테스트를 하기도 편한 코드를 만드는 것을 점검하는 것도 좋은 방법이다. 물론, 테스트가 쉬운 코드들은 대부분의 경우 구조적으로도 우수한 경향이 있다.

- 01. 코드가 읽기 쉽게 작성되어 있는가?**
- 02. 코드가 코딩 룰이나 스타일을 일관되게 따르고 있는가?**
- 03. 코드가 구조화 되어 있으며 지나치게 복잡하거나 긴 부분이 없는가?**
- 04. 코드가 문법적으로는 맞지만, 오류가 날 가능성은 가진 부분은 없는가?**
- 05. 코드가 적절하게 배치 되어 있는가?**
- 06. 코드가 하려는 일과 코딩 내용이 일치하는가?**
- 07. 코드의 길이가 적절한 크기인가?**
- 08. 코드에 하드 코딩 된 부분이나 필요없는 부분은 없는가?**
- 09. 오류가 발생했을 때 어떻게 회복하는가?**
- 10. 복귀값 및 입력값이 있을 경우 그것들을 점검 하는가?**

이상과 같이 대략 10가지를 적어보았지만, 이것은 단지 참고로 나열한 것일 뿐이다. 프로젝트의 상황이나 팀의 역량에 맞게 수정하거나 더 추가해도 상관없다. 코드 리뷰를 아무런 원칙이 없이 하는 것이 아니라면, 충분히 설득력을 가진 점검 항목으로 구성하면 된다. 구성된 내용에 따라 점검하고, 그것을 코드 리뷰의 로그로 남기면 될 것이다. 이때, 문제의 양상이나 영향도와 같은 부분도 기록하면 좋을 것이다. 나중에 남겨진 로그를 가지고 분석한 데이터를 이용해서, 자주 나타나는 잘못된 코딩 습관을 교정할 수 있는 자료로 활용할 수 있을 것이다. 코드 리뷰는 선택이 아니라 습관이어야 하며, 그것은 객관화된 코드를 바라보는 관점과 코드의 공동소유를 추구한다는 것을 나타내는 활동이어야 한다.

[변수(Variable)에 대해]

변수는 변화하는 수를 넣어두기 위한 공간이다. 변수는 넣어 둘 수 있는 종류와 값을 한정적으로 가지고 있으며, 이를 형(Type)이라고 부른다. 따라서, 같은 타입이라면 변수에 같은 값을 저장할 수 있다. 다른 타입을 저장하기 위해서는 타입의 변환이 발생하게 되며, 이때 정보의 손실이 발생할 가능성이 있다. 즉, 변수는 자신이 표현할 수 있는 값의 범위를 가지고 있으며, 그 범위를 넘어서거나 표현할 방법을 제대로 찾기 어려운 경우, 정보의 손실을 동반한 가장 가까운 형태로 저장할 값을 변환한다.

또한, 변수는 형식만 지정하는 것이 아니라 크기도 가지고 있다. 크기에 따라 다른 변수로 선언해 줄 필요가 있다. 대체로 더 큰 타입의 변수로 변환할 때는 정보의 손실이 발생하지 않지만, 더 작은 타입 이거나 저장할 수 있는 값의 변화가 심할 경우에는 정보손실을 일으킨다. 대부분의 경우 변수는 정해진 형태로 사용 되는 것이 일반적이며, 크기가 다른 타입간에는 변환의 비용이 따를 수도 있다. 따라서, 될 수 있으면 변수의 타입을 선언한 이후에 다른 타입으로 변경하는 것은 좋은 선택이 아니다.

변수를 사용하는데 있어서 가장 주의해야 할 부분은 변수의 이름을 만드는 일이다. 잘못된 이름을 가지게 되면, 코드 전반에 대한 이해를 어렵게 만들기도 한다. 좋은 변수는 묘사적인 이름을 가지며, 누가봐도 쉽게 이해할 수 있는 이름을 가지고 있다.

```
int temp1, temp2;
```

위와 같은 이름을 주는 경우를 흔히 발견할 수 있다. 물론, 나쁘다는 것은 아니지만, 이 변수의 이름으로 변수가 하는 일을 추정하기는 어렵다는 점은 이야기하고 싶다. 이 변수가 사용되는 범위는 "temp"라는 말에서 알 수 있듯이, 잠시 동안이라고 생각할 수 있다. 하지만, 코드의 수정이 발생하면 생각보다 잠시 동안은 긴 시간이 될 수 있다. 좀 더 묘사적인 이름을 준다면, "old_age"와 같이 명확하게 주는 것이 좋다. 즉, 잠시 동안 사용할 변수라고 생각하더라도, 코드의 이해를 돋는 방법이 이해하기 쉬운 코드를 만든다.

물론, "for()"문과 같은 곳에서 사용하는 변수들은 "i, j, k"와 같은 것을 사용해도 된다. 이런 경우에는 이미 개발자들이 마치 일반화 된 규칙처럼 적용하고 있기 때문이다. 따라서, 그런 경우라면 일반화된 관용적인 표현을 사용하는 것이 오히려 코드를 이해하기 쉽게 만들어준다. 하지만, 코드에서 사용하는 다른 변수들은 이름을 가지는 것이 훨씬 더 읽기에 도움이 된다.

코드에는 변수는 아니지만 상수로 사용되는 값들도 많다. "Magic Number"라고 부르는 마법의 숫자들은 코드를 정말 이해하기 어렵게 만드는 경향이 있다. 이런 것들도 이름을 지니고 있는 것이 읽기에 유리하다. 나중에 코드를 관리할 때도 물론 이점을 가진다. 숫자로 만들어진 값을 그냥 적어두면 나중에 그것들이 왜 거기에 있는지, 혹은 그것이 다른 코드 부분에서 정의한 값과 같은 것인지 다른 것인지를 구분하지 못하게 된다.

```
int x = 256;
...
if ( k != 256 ) {
    ...
}
```

위와 같은 코드를 만나면, 사람들은 "256"이라는 숫자가 왜 나왔는지 궁금하게 된다. 하지만, 그것을 대답해줄 단서는 코드에서 찾기가 힘들며, 그것을 물어볼 사람이 바쁘다면 다른 값으로 고치기가 난처하다. 따라서, 이런 코드는 수정을 기피하게 만들며, 문제를 키우는 코드로 이어질 가능성이 높다. 차라리, "#define MAX_NAME_SIZE 256"과 같이 정의해 두고, "MAX_NAME_SIZE"를 이용하면, 값의 의미를 더 명확히하고 그 값이 사용된 코드들의 상관 관계를 명확히 파악할 수 있을 것이다.

변수는 사용되기 전에 선언되어야 한다. 즉, 컴파일러가 변수의 크기와 주소를 판단하기 위해서 사용할 정보를 주어야 한다. 하지만, 문제는 그럼 언제 그것을 선언하는 것이 좋은가로 이어지게 된다. 가장 좋은 것은 사용되기 바로 전에 선언하는 것이다. 기존의 C문법에서는 무조건 함수의 내부에서 첫 부분에 변수가 선언되어 있어야 했지만, "C99"표준에서는 이런 규칙이 완화 되었다. 따라서, 좀 더 변수의 유효 범위를 한정하는 것이 가능해졌다. 변수는 가능한 사용되는 범위를 줄이는 것이 버그 발생을 줄여준다.

```
...
for( int i; i < MAX_NAME_SIZE; i++ ) {
    ...
}
```

위와 같은 'for()' 반복문의 경우, 변수 i의 범위는 "for()" 반복문의 범위를 넘어서지 못한다. 즉, "for()" 문 안에서만 의미를 가진다. 따라서, 변수와 관련된 문제가 발생할 가능성도 줄어든다. 사용 범위가 넓어지면 문제의 범위도 같이 넓어진다. 코드가 길어지면 코드의 문제 범위가 넓어지듯이, 변수도 코드의 일부 이기에 한정된 사용범위를 가지는 것은 중요하다.

전역 변수는 전역적인 문제를 발생시킨다. 따라서, 전역 변수는 가능한 쓰지 않는 것이 좋다. 만약, 써야 할 필요가 있다면, 한정적으로 정의된 부분에서만 사용하는 것이 좋다. 구현의 자유도가 높으면, 문제의 발생 가능성도 같이 커진다. 따라서, 전역 변수를 한정적으로 사용해서 한정적인 부분에서만 문제가 있도록 만들면(혹은, 그 부분만 제대로 챙기면) 코드의 관리면에서 효율적이다. 문제의 범위를 좁혀가는 것이 모든 문제의 해결책 이듯, 다양한 부분에서 생길 문제를 한정적으로 다루는 기술은 여기서도 필요하다.

각 변수의 이름에 타입이나 크기를 명시하는 것이 필요할까? 최근의 분위기는 그런 것들을 안붙이는 방향으로 흘러가고 있다. 그런 표기를 만들어 냈던 사람조차도 그런 의도에서 만든 것이 아니라고 이야기하고 있다. 즉, 변수의 타입과 크기는 컴파일러에서 관리하는 수준에 있어야 하지, 사람이 관리할 것이 아니다.

```
char *gpzsName = "My Name"; /* Global Pointer to Zero Terminated String */
```

위의 코드는 "g"를 이용해서 전연(Global) 변수임을, "pz"를 이용해서 "Zero로 끝나는 문자열을 가르키는 포인터 임"을 나타냈다. 하지만, 이것이 코딩에 있어서 큰 도움을 줄 수 있을까? 그렇지 않다고 본다. 그냥, "char *name"이라고 해도, 컴파일러는 제대로 판단해서 오류가 나는 것을 찾아낼 수 있다. 오히려 "gpz"와 같은 것을 사용하는 노력만 더 추가될 뿐이다. 그럴 시간이 있다면 좀더 변수를 묘사적으로 만드는데 투자하는 편이 코드의 이해를 쉽게 만들 것이다.

전역 변수를 사용해야 하는 경우라면, 전역 변수에 접근자를 만드는 것이 도움이 될 것이다. 물론, 이것을 오버헤드(Overhead)로 보는 견해도 있다. 즉, 변수를 직접 접근해서 사용하는 것이 여려모로 성능에 도움이 된다고 보는 것이다. 만약, 함수를 이용해서 접근자를 만들경우, 이렇게 해서 줄 일 수 있는 오버헤드는 함수의 호출과 복귀에 따른 CPU사이클의 소모와 변수의 메모리 공간에 접근하기 위한 메모리 접근 사이클일 것이다. 요지는 작은 개선이 있다는 것 뿐이다.

전역 변수 자체를 직접 사용하더라도 접근에 필요한 메모리 사이클은 필요하다. 즉, 전역 변수들은 레지스터에 할당받지 못하고 접근할 때마다 메모리에서 읽어와야 한다. 따라서, 남는 것은 함수의 호출 오버헤드다. 반복문과 같은 곳이 아니라면, 이것도 큰 틀에서 보면 심각한 부분은 못된다. 정말 필요한 부분은 전역 변수의 남용으로 발생하는 시스템 전체의 파급효과이지 몇 번의 함수 호출이 아니다. 따라서, 가능한 전역 변수의 남용을 줄이는 것이 더 좋은 성능으로 이어질 가능성이 높다.

전역 변수를 줄이기 위해서는 최대한 변수의 사용 범위를 줄이는 것이 좋다. 즉, 시스템의 구조를 역할과 책임에 따라 제대로 구분해 주는 것이 선행되어야 한다. 전역 변수는 접근하는 이유는 변수가 정의된 곳과 사용 되는 곳 사이가 시간이나 공간적으로 차이가 있기 때문이며, 그것을 가능한 모아 주면 역할과 책임의 범위가 명확해 진다. 물론, 그렇다고 완전히 전역변수를 다 제거할 수 있는 것은 아니다. 최대한 줄일 수 있는데 까지 줄여주어야 한다.

헤더 파일과 같은 곳에는 당연히 변수(특히, 자료구조를 표현하고 있는)에 대한 구체적인 내부 구현 내용까지는 포함되지 않아야 한다. 그것을 포함시키는 순간, 그것에 대한 책임과 역할은 분리가 되기 때문이다. 즉, 다양한 코드들이 내부 구조를 이용해서 손쉽게 변수를 변경시킬 수 있게 된다. 이런 변경이 발생하면 그 책임을 정하기 어려워진다. 따라서, 변수는 선언된 곳에서 책임을 지는 것이 일반적이다. 자료구조가 정의 된 곳에 해당 자료구조를 다루는 함수들을 두는 것이 이러한 이유 때문이다. 다른 자료구조를 많이 접근해야 한다면, 그 함수 자체를 원래의 자료구조가 사용 되는 곳으로 이동시키는 것이 바람직하다.

[변수의 타입에 대해서]

C언어에서 변수는 타입을 가진다. 변수가 타입을 가지는 이유는 값을 저장하는 방식과 사용하는 방식에 제약이 생기며, 부주의한 사용은 프로그래머가 의도한데로 프로그램이 동작하지 못하게 만들 수 있다. 예를 들어, 같은 정수 타입이라고 하더라도, 부호가 있는 정수(Signed integer)와 부호가 없는 정수(Unsigned Integer)는 다른 타입이다. 따라서, 만약 이것들을 섞어서 사용한다면, "의미있는 값"을 잃어버릴 위험이 발생할 수 있다는 뜻이다. 코딩 룰 표준인 MISRA-C 2012는 이런 것들도 기계적으로 검사해서 개발자의 실수를 최소화는데 도움을 준다.

비트(Bit) 연산과 관련해서도 변수의 타입을 신중하게 사용해야 한다. 개발자의 실수를 최소화하기 위해서는 비트 관련 연산에서 사용하지 못하는 데이터 타입이나 섞어쓰지 말아야 하는 데이터 타입을 알아야 한다. 만약, 부동 소수(Float Point)를 비트 관련 연산에서 사용한다면 어떤 일이 발생할까? 아마도 변수를 표현하기 위해서 사용하는 비트 필드들이 엉뚱한 조작을 통해서 원하는 결과를 얻기는 어려울 것이다. 혹은, 컴파일러에 의해서 경고 메시지가 출력될지도 모른다. 컴파일러가 알려주는 경고들을 무시하거나 컴파일 옵션에서 끈 상태라면 이런 것들에 대한 대비책은 아무것도 없을 것이다.

포인터의 사용도 정의된 타입만 한정적으로 이용해야 한다. 포인터가 하나의 타입에 대해서 선언된 것을 타입 캐스팅된 다른 타입에 대해서 사용하는 경우, 경고는 없을 지라도 그 변수가 가져야 하는 값의 범위와 부호에 대해서 영향을 줄 수 있다. 따라서, 하나의 타입에 대해서 선언된 포인터를 다른 타입으로 정의된 변수에 사용해선 안된다. 이런 것들이 당장은 문제를 일으키지 않더라도 잠재적인 오류가 될 수 있다. 특정 상황이되면(변수가 그리키는 최대 혹은 최소값에 가까운 연산을 하게되는 경우) 문제를 일으키고, 그 원인은 찾기 힘들 가능성이 높다.

```
int array[ 100 ];
int *pointer = array;
*(pointer + 10) = 1000; /* Easy to make an error */
pointer[ 10 ] = 1000; /* Easy to understand and less error prone */
```

포인터를 배열을 처리하기 위해서 사용할 때는 특히 주의해야 한다. 자주 문제가 발생하는 부분은 포인터가 배열의 경계를 넘어설 때다. 따라서, 이를 해결하기 위해서는 마치 포인터를 배열처럼 사용하는 것도 한 가지 방법일 수 있다. 예를 들어, "int *pointer = a[100];" 이라고 정의했다면, "p = a; p[5] = 100;"과 같이 포인터가 아닌 배열처럼 사용하는 것이다. 문법이 사용할 수 있는 모든 경우를 활용하도록 요구할 수 있지만, 그렇다고 지나치게 문법에 치중하기보다는 오류가 발생할 가능성이 가능한 적게 코딩하는 것이 유리하다. 포인터는 C언어에서 가장 이해하기 어려운 부분이며, 개발자들이 실수를 하기 쉽

기 때문에 한정적으로 사용하도록 유도해야 한다.

변수의 타입을 캐스팅 해서 사용하는 것은 좋지 않다. 즉, 변수가 가지고 있는 값에 영향을 줄 수 있기 때문이다(복귀 값으로 사용되는 변수도 마찬가지다). 어쩔 수 없이 그렇게 해야 한다고 생각하기 보다는, 차라리 새로운 타입을 정의해서 쓰는 것을 고려해 봐야 할 것이다. 기존에 유사한 타입이 있다고, 그것을 확장하는 의미에서 "union"과 같은 것을 사용하는 것도 줄여야 한다. "union"은 변수의 타입을 일관성 없게 다룰 수 있기 때문에 혼동을 가져올 가능성이 높다. 구조체를 선언해서 "bit field"별로 이름을 주는 것도 흔히 사용하는 방법이지만, "bit field"의 크기를 넘어서지 않도록 주의해야 한다. 그렇지 않다면 차라리 합쳐서 하나의 이름을 주고, 그 이름의 변수를 "bit mask"를 이용해서 사용하는 것도 실수를 줄일 수 있다.

변수의 타입은 중요한 요소이지만, 제대로 신경 쓰지 않는 경우가 흔하다. 개발자들은 변수를 처음 선언할 때, "대략적으로 이 정도면 되겠지"라고 생각해서 정의한다. 하지만, 나중에 문제가 발생하면 생각보다 디버깅하는 노력이 많이 들어갈 가능성이 높다. 특히, 부호가 있는 정수와 그렇지 않은 정수를 다루는데는 일관성이 필요하지만, 별로 개의치 않는 것도 사실이다. 물론, 이런 부분들을 일일이 개발자가 점검하는 것은 힘든 일이다. 따라서, 우리는 도구를 이용해서 개발자가 놓칠 수 있는 사소한 부분들을 기계적으로 관리하는 도구를 사용하는 것이다. 도구에 지불하는 비용이 비싸다고 생각해선 안된다. 도구가 가져다 주는 이점에 비교했을 때 정말 비싼 경우는 드물다. 좋은 도구는 개발자의 노력을 좀 더 가치있는 일에 집중하도록 만들어 준다.

[암시적인 타입 변환 보다는 명시적인 타입 변환을 사용]

타입 선언이 필요없는 스크립트 언어인 경우에는 내부적으로 타입을 처리하는 일이 자동으로 일어나지만(일반적으로 변수의 값은 따로 저장하고, 저장된 곳을 가르키는 포인터를 사용한다), 타입 선언을 가지는 언어는 정확히 타입을 사용하는 것이 오류를 줄여준다. 변수의 타입은 가질 수 있는 값의 범위를 지정하기에, 그 이상의 값을 가지는 경우가 발생하면 오류를 유발할 가능성이 높다. 따라서, 암시적으로 (Implicit)으로 타입을 변환시켜주는 오류를 내포할 가능성을 항상 가지고 있는 것이다.

```
signed char a = 255;
unsigned char b = a;

printf("The value : %d, %d\n", a, b);
```

위와 같은 코드가 있다고 했을 때, 결과값은 어떻게 될까? 물론, 이런 코드를 작성하는 경우는 극히 드물 것이다. 가까운 위치에 정해진 변수들을 사용하고 있기에, 타입간의 호환이 안된다는 것도 쉽게 눈치챌 수 있을 것이고 고치는 것도 쉽다. 하지만, 둘 간에 정의된 위치가 달라질 경우에는 의도하지 않은 결과가 나올 수 있다. 위의 코드를 실행 했을 때 얻는 값은 각각 "-1, 255"라는 값이다. 아마도 개발자는 두 개의 변수가 같은 값을 가져야 할 것이라고 생각했을지는 모르지만, 다른 값이 나왔다는 것을 알 수 있다.

```
b = (unsigned char)a;
```

위의 코드는 명확히 코드의 의미를 알고 사용한 경우다. 즉, "a"라는 변수에 들어있는 값을 "b"라는 변수에 저장할 때, "unsigned char"형태로 변환 될 것이라는 것을 알고 있는 것이다. C언어에서는 위와 같은 변수의 타입 변환에 대해서 경고 메시지를 출력하지 않는 경향이 있으며, 이는 실수가 발생할 가능성을 열어두고 있는 것이다. 따라서, 이미 알고 있는 것을 미리 이렇게 표현해주는 것은 잘못된 값을 받아들일 가능성을 배제할 수 있는 좋은 방법이다. 하지만, 이것도 궁극적인 해결책은 아니다.

```
unsigned int a = 65536;
unsigned int b = a;
```

가장 좋은 방법은 타입이 같은 것들만 함께 사용하는 것이다. 즉, 타입이 달라서 생기는 문제를 사전에 미리 제거하는 것이다. C언어에서는 타입간의 변환을 암시적으로 컴파일러가 실행하기에, 개발자가 모르는 사이에 의도와는 다른 값이 사용되는 경우가 종종 있다. 따라서, 이런 경우를 사전에 제거하는 것이 가능한 버그를 줄이는 방법이며, 같은 타입끼리 연산을 하는 것이 그런 가능성을 최대한 막아준다. 물론, 항상 같은 타입만으로 코딩하는 것이 불가능한 경우도 있을 것이다. 그럴 때는 인위적인 타입 변화를 명시적(Explicit)으로 사용하는 것이 코드를 보는 사람에게 명확한 의도를 전달하는 방법이다.

"MISRA-C 2012"와 같은 코딩 룰(Rule)을 검사하는 툴에서는 암시적인 타입 변환에 대해서 경고를 주고 있기에, 도구를 활용해서 잘못된 부분을 찾아내서 고칠 수 있다. 물론, 이런 것들이 너무 제약이 심하다고 말하는 개발자들이 많지만, 사소한 오류들에 대해서 개발자가 집중하기보다는 그런 것들은 도구에서 걸리지도록 만들고, 개발자는 좀 더 어려운 문제를 찾는 것이 효율적이라고 본다. 이런 관점에서 개발에 관련된 도구에 대한 투자는 상대적으로 오히려 비용이 싸다고 생각되기도 한다.

```
int k = -1;
unsigned int j = k;
int i = j + k;

printf("The value : %d, %d\n", k, j);
printf("The value : %d, %u\n", k, j);
j++;
printf("=====\n");
printf("The value : %d, %d\n", k, j);
printf("The value : %d, %u\n", k, j);
printf("The value : %d\n", i);

.....
.....
.....



int16_t k = -1;
uint16_t j = k;
int16_t i = j + k;

printf("The value : %d, %d\n", k, j);
printf("The value : %d, %u\n", k, j);
j++;
printf("=====\n");
printf("The value : %d, %d\n", k, j);
printf("The value : %d, %u\n", k, j);
printf("The value : %d\n", i);
```

참고로, 위의 코드는 "stdint.h"를 사용한 정수 변수의 선언을 보여주는 예다. 여기서, 주의할 점은 거의 동일한 코드로 보이지만, 결과가 다르다는 점과 "unsigned int"와 "signed int" 간의 덧셈의 결과가 우리가 예상하는 결과와 달라질 수 있다. 각각을 실행한 결과는 아래와 같다.(코드는 윈도우에 설치된 Eclipse Mars버전에서 MinGW를 이용해서 컴파일 되었다.)

(첫번째 코드)

```
The value : -1, -1
The value : -1, 4294967295
=====
The value : -1, 0
The value : -1, 0
```

The value : -2

(두번째 코드)

The value : -1, 65535

The value : -1, 65535

The value : -1, 0

The value : -1, 0

The value : -2

결과에서 보듯이, 첫 번째 코드의 "printf()"문의 결과가 다르다는 것을 확인할 수 있다. 또한, "unsigned int"값이 오버플로우(Overflow)가 발생한 상황에서는 동일한 결과를 보이지만, "unsigned int"와 "signed int"간의 덧셈이 예상 결과 값과 다르다는 것도 확인할 수 있다. 따라서, 여기서 이야기 할 수 있는 부분은 될 수 있으면 동일한 타입으로 연산을 적용하는 것이 좋다는 점과, "stdint.h"와 같이 정수 타입을 정의하는 헤더 파일에 정의된 타입을 사용하는 것이 조금 더 프로그래머의 의도에 가깝게 코딩할 수 있다는 것이다.

```
int16_t k = -1;
uint16_t j = (uint16_t) k;
uint16_t i = j + k;
```

아마도, 개발자의 의도는 위의 코드와 같이 사용하는 것을 원했을지도 모른다. 즉, 변수의 타입을 명시적으로 변화하고, 연산의 결과를 저장하기 위해서도 동일한 타입으로 저장 하기를 원했을 것이다. 의도가 어떻든 컴파일러의 입장에서는 그것을 알아낼 방법이 없기 때문에, 가능한 컴파일러가 적절히 대응할 수 있도록 충분히 “방어적으로 코딩”하는 것을 선택해야 할 것이다. 실행 결과를 아래와 같다.

The value : -1, 65535

The value : -1, 65535

The value : -1, 0

The value : -1, 0

The value : 65534

[타입 캐스팅(Type Casting)을 주의]

타입 캐스팅이 필요한 경우도 있지만, 될 수 있으면 인위적인 변환을 사용하지 않는 것이 좋다. 변수가 선언된 타입을 그대로 사용하는 것이 최대한 오류를 줄이는 방법이다. 물론, 할당된 메모리와 같은 경우에는 타입캐스팅을 해야 하는 것이 기본이다. 전달받은 데이터의 구조가 정해지지 않은 경우도에 어쩔 수 없이 사용해야 한다. 그런 몇 가지 경우를 제외한다면, 변수나 자료형에 맞게 사용하는 것이 오류를 줄여줄 것이다.

```
int x;
float y;
```

```
x = (int) y;
```

예를 들어, 위의 코드와 같은 경우에는 "float"로 선언된 "y"변수의 값이 "x"에 할당되면서 유효한 값을 잃어 버릴 가능성이 있다. 그리고, 같은 타입이라고 하더라도 크기가 다른 경우(예를 들어, uint8_t과 uint16_t은 변수의 값을 저장하는 크기가 다르다.)도 마찬가지다. 물론, 암시적으로 "x = y"와 같이 해도 유효한 값을 잃어버리는 것은 당연하다. 요점은 변수의 크기를 충분히 확보하고, 그것에 맞는 타입으로

연산에 적용하라는 말이다.

변수의 타입을 잘못 사용하는 경우는 특정 변수의 값을 임의로 선택하는 오류를 범하기 쉽다. 그것을 방지하기 위해서 포팅(Porting)을 위해 자신만의 타입을 정의해서 사용하는 경우도 있으며, 특별한 변수의 이름을 주는 규칙을 만들기도 한다. 하지만, 타입을 점검하는 것은 컴파일러의 몫이다. 컴파일러를 일단 신뢰하는 것부터 시작해야 할 것이다. 예를 들어, 컴파일러에서 "잘못된 변수 변환"이 있다는 경고 메시지가 있다면 그것 부터 제거하는 것이 옳다.

예전에는 "gui8_myValue"과 같이 변수의 이름에 변수의 범위(scope)와 타입을 명시적으로 적어두는 것을 코딩룰로 하는 경우가 많았다. 최근의 경향은 그냥 변수의 이름을 잘 주는 방향으로 바뀌었다. 하지만, 아직도 이런 분위기 많이 살아 있으며, 다양한 책들에서도 예제로 많이 발견되고 있다. 사실 이것은 별 필요가 없는 일이기도 하다. 이름의 범위에 문제가 발생할 수 있는 경우는 중복된 정의와 같은 경우이며, 컴파일러가 그런 것들을 찾아낸다. 물론, 정의하지 않고 전역 변수를 접근하는 경우에는 컴파일 오류가 발생한다.

```
#include <stdint.h >

unsigned short gui8_myValue; /* Global Unsigned Integer */

.....
uint8_t myValue;
```

최근의 경향은 그냥 이름을 더 잘 짓는 것과 컴파일러의 기능을 충분히 이용하는 것이다. 그리고, 가능한 전역변수의 사용을 막고, 변수의 범위를 좁혀서 사용하는 것이다. 예를 들어, 블록("{ }")내에서만 필요한 변수는 해당 블록에서만 정의해서 사용하는 형태다. 전역적인 자료구조의 사용이 필요한 경우도 있지만, 그것 마저도 다른 인터페이스로 대체가 가능하다. 따라서, 변수의 이름에 타입이나 크기와 같은 정보를 붙여주는 것은 별로 좋은 선택이 아니다. 크기의 경우에는 오버플로우(Overflow)가 발생하는 경우가 있다. 하지만, 그렇다고 크기 정보를 변수의 이름에 명시한다고 발생하지 않는 것은 아니다.

[함수(Function)에 대해]

C언어는 가장 작은 실행 단위가 함수로 만들어지는 언어다. 함수의 크기와 역할은 다양하지만, 기본적으로 입력에 대한 데이터의 조작과 출력으로 이뤄진다. 흔히 " $y=f(x)$ "와 같이 함수가 기술되듯이, 함수의 하는 역할을 대표하는 " $f()$ "가 무엇인지를 파악할 수 있다면, 출력 " y "는 추측이 가능하다. 즉, 함수가 하는 역할은 " $f(x)$ "가 표현해 줄 수 있어야 한다.

함수의 이름도 변수의 이름과 마찬가지로 묘사적으로 정의되어야 한다. 무슨 일을 하는지 함수의 이름만 보고도 파악할 수 있다면, 함수를 정확하게 이해하려는 노력은 줄어들 것이다. 대부분의 오류가 함수와 함수 사이에서 발생한다는 것은 함수간에 서로 약속된 일을 하지 않았거나, 혹은 잘못된 이해를 가지고 함수를 만들었다는 말이 된다. 이런 것을 일종의 “계약(Contract)”이라고 본다면, 계약의 불이행이 오류의 원인이라고 할 수 있다.

함수의 이름이 묘사적(Descriptive)이어야 한다고 반드시 길게 만들어야 하는 것은 아니다. 그리고, 특정 함수의 이름에는 그 함수가 속한 모듈의 이름이 나올 수도 있다(임베디드 시스템과 같은 경우 이런 규약을 많이 사용한다. 예를 들어, “NW_send_packet()”과 같이 네트워크 관련 모듈의 “send_packet()” 함수). 어떤 모듈에 정의되어 있는지를 정하는 것은 함수의 책임과 범위를 명시하기 위한 것이며, 그것으로 인해서 영향을 받을 수 있는 부분을 명확히 할 수 있어서 도움이 되기도 한다. 함수의 이름에 약자를 사용하는 경우도 있는데, 이 때는 팀원들이 공통으로 동의하고 업계에서 당연하게 사용되는 용어인 경우

에 해당한다. 따라서, 될 수 있으면 잘 모르는 단어나, 혼자만 알고 있는 약자는 쓰지 않는 것이 좋다.

```
char *MM_Allocate_Buffer( const size_t size );
```

예를 들어, 위의 코드는 MM(Memory Manager)라는 모듈에 버퍼를 할당하는 함수를 정의한 것이다. 함수 정의는 사용하는 측에서 명확히 이해할 수 있도록 정보를 주어야 하며, 최대한 많은 정보를 간략하게 표현할 수 있어야 한다. 입력으로 주어지는 파라미터의 변경이 생기지 않는다면, "const"와 같은 것을 사용해 주는 것이 좋다. 물론, 이것으로 함수를 완벽하게 이해할 수 있다고는 장담하지 못한다. 따라서, 필요한 것은 함수에 대한 설명이 추가될 수 있어야 한다.

이때 사용할 수 있는 방법으로는 "Doxygen"과 같은 프로그램을 이용하기 위해서, 특정 형식으로 주석을 추가하는 방법이 있으며, 직접 사용되는 예제를 만드는 방법도 있을 것이다. 경험상 둘 다 사용하는 것이 문서화 및 개발자에게 직접적인 도움을 준다고 생각한다. 예제를 만드는 방법은 간간한 단위 테스트를 만드는 것으로 대체할 수 있다. 즉, "단위 테스트(Unit Test)"에서 직접 해당 함수를 사용해서 어떻게 동작하는지를 파악할 수 있기 때문이다. 이처럼 단위 테스트는 문서화를 일부 대체할 수 있는 역할도 해줄 수 있다.

```
/****************************************************************************
 * @name MM_Allocate_Buffer( const size_t size )
 * @description This function will allocate one buffer specified in size
 * @param size The size of the buffer in bytes which is allocated
 * @return The allocated buffer address pointer
 */
char *MM_Allocate_Buffer( const size_t size );
```

위의 코드는 간단한 "Doxygen" 주석 포맷의 사용을 보여준다. Doxygen은 일반적으로 많이 사용하는 프로그램이므로, 시간이 있을 때 잠시 읽혀두는 것이 좋을 것이다. 주석으로 코드를 설명하는 것의 단점은, 코드는 수정되는데 주석은 잘 바뀌지 않는다는 점이다. 따라서, 코드의 수정이 발생했을 때, 반드시 주석도 수정되었는지 코드 리뷰에서 확인해주어야 한다.

함수가 가지는 파라미터의 수는 적을 수록 좋다. 하나도 없다면 가장 좋을 것이다. 될 수 있으면 복귀 값은 있는 것이 좋지만, 반드시 있어야 할 필요는 없다. 단위 테스트에서 검증을 위한 방법으로 내부에서 사용하는 자료구조를 외부에서 파라미터로 입력하거나(의존성 삽입(Dependency Injection)), 파일 범위의 변수를 넣을 수도 있다. 하지만, 복귀값을 이용하는 것이 단위 테스트에서 일반적으로 사용하는 방법이기에, 복귀 값으로 함수가 정확히 구현 되었는지 확인할 수 있도록 만들면 될 것이다. 함수의 파라미터의 갯수는 최대 4개를 넘지 않도록 하는 것이 일반적으로 좋은 성능을 보인다(컴파일러가 4개까지는 레지스터를 이용한 파라미터 전달).

```
TEST_ASSERT_EQUAL( NULL, MM_Allocate_Buffer( 0 ) );
TEST_ASSERT_NOT_EQUAL( NULL, MM_Allocate_Buffer( 2 ) );
...
```

함수의 길이는 될 수 있으면 짧아야 한다. 그리고, 함수는 잘 정의된 한 가지 일만 처리해야 한다. 즉, 함수의 이름이 나타내는 일만 해야한다. 함수의 이름에 "xxx_and_yyy"와 같은 것이 있다면, 이는 함수가 두 가지 이상의 일을 할 수 있다고 정의한 것이다. 이런 것은 좋은 이름이 아니며, 그 구현도 길어질 가능성이 높다. 함수는 잘 정의된 한 가지 일만 할 경우 재사용될 가능성이 높다. 한번에 많은 일을 하려고 하면, 코드는 길어지고 복잡하게 될 가능성이 높다. 대략적으로 모든 함수는 100라인 이하로 만드는 것이 좋다(더 작아도 좋다). 그 이상의 크기를 가지거나 여러가지 일을 처리한다고 생각되면, 함수를 쪼개는

것이 도움이 된다.

함수의 위치도 문제다. 만약, 사용하는 자료구조가 다른 모듈에 속한 것이거나, 다른 모듈의 자료구조를 지나치게 많이 참조하고 있다면, 그 함수의 위치는 잘못된 것이다. 따라서, 해당 자료구조가 있는 모듈로 옮겨주는 것이 좋다. 함수간의 호출이 여러 모듈을 거칠 경우에는 함수가 거쳐가는 모듈들의 의존성이 순환(Cycle) 의존 관계를 만들지 않는지 면밀하게 검토해야 한다. 이런 순화적인 의존성이 발생하면, 코드들은 군집해서 다루어야 하는 상황이 발생한다. 즉, 물리적으로는 나누어진 것처럼 보이지만, 논리적으로 큰 덩어리를 생성하기 때문이다.

함수의 이름은 자신이 하는 일을 대표한다. 따라서, 하나의 함수 내부에 있는 코드들은 함수 이름보다 한 단계 낮은 수준의 추상화로 이뤄지는 것이 좋다. 즉, 함수 내부의 코드들이 지나치게 논리적인 구체화나 비약이 있으면 코드를 이해하는데 어려움을 줄 수 있다. 또한, 구체화와 추상화가 혼재 되면, 나중에 코드를 계층적으로 분리하는데도 어려움을 겪게 된다. 물론, 계층을 많이 가져가는 것은 함수 호출의 오버헤드를 증가시킬 수 있다는 점도 생각해 봐야 하지만, 코딩을 시작할 때는 이런 것들은 나중으로 미루고 계층화를 시키는 것이 좋다. 최적화된 코드는 구조화를 위해서 변경하지 못하지만, 구조화된 코드는 최적화를 위해서 쉽게 변경이 가능하기 때문이다.

```
char *MM_Allocate_Buffer( size_t size ) {
    if ( size == 0 ){
        return NULL;
    }
    return (char*)malloc( size );
}
```

함수 내부의 지역변수의 개수도 아껴서 사용하는 것이 좋다. 레지스터 수준에서 지역 변수를 다루지 않으면 함수의 실행 속도는 낮아진다. 즉, 메모리나 캐쉬를 접근하는데는 CPU의 명령어를 실행하는 속도 보다 항상 더 많은 비용이 들어가기 때문이다. 이런 일이 가능하기 위해서는 당연히 함수의 길이가 커서는 안된다는 것을 알 수 있을 것이다. 길어지는 만큼 사용해야하는 자원(변수)도 늘어나게 된다.

함수 내부의 코드들은 지나치게 중첩되어 사용되지 말아야 한다. 조건문이나 반복문 등은 중첩해서 사용 할 수 있지만, 중첩이 발생하면 코드의 이해는 더 어려워진다. 중첩을 발생시키지 않기 위해서는 함수의 진입점(Entry Point)은 한 곳이지만, 진출점(Exit Point)은 여러개를 가질 수도 있다. 이것에 대해서는 논란이 있지만, 여기서는 단순히 여러 개의 진출점(return 문을 사용하는 것)을 가지는 것을 권장하도록 하겠다. 이유는 코드를 좀 더 중첩이 없는 평탄한(flat) 구조로 만들 수 있기 때문이다. 평탄한 구조는 화이트 박스(While Box) 테스트에서 조건을 명시하는 것을 편하게 해줄 수도 있다. 단점으로는 할당된 자원이 있을 경우에는 코드의 진행 상황에 맞게 주의해서 해제를 해야 한다는 것이다.

[함수의 출구는 몇개가 좋을까?]

함수는 적은 수의 입력을 받아서, 하나의 출력을 만들어내는 것이 좋다. 입력의 경우의 수가 적을수록 해야 할 일이 명확해지고, 테스트 해야 할 입력 값의 조합도 간단해 진다. 출력은 하나인 것이 좋다. 여러 가지를 출력으로 보낸다면(함수의 복귀값과, 포인터 파라미터를 이용하는 경우), 상태의 변경과 출력을 동시에 하는 함수가 만들어지기 때문이다. 이때는 함수를 상태를 변경하는 것과 상태를 파악하는 것으로 나누어 구현하는 것이 좋다.

그렇다면, 함수의 출구는 몇 개를 가지는 것이 좋을까? 함수의 출구를 하나로 만드는 방법은 이미 많이 이야기 되어 왔다. 즉, 구조적인 프로그래밍 기법의 하나로 단일 출구를 만드는 방법은 존재한다. 하지만, 이렇게 할 경우 제어문이 복잡해지는 경향이 생기게 되며, 코드를 이해하기 어렵게 만들기도 한다. 따라서, 읽기 측면에서 편하도록 만들려면 여러 개의 출구를 함수가 가질 수 있도록 해야한다.

```
int function(int x, int y) {
    int result = 0;

    if (x > 0) {
        result = x + y;
        if (y <= 0) {
            result += x * y;
        }
    }
    return result;
}
```

위의 코드는 두 개의 "if()" 문이 중첩되어 있다. 각각의 조건에 따라, 결과 값(result)을 계산하는 방식은 차이를 보이고 있다. 이를 좀더 이해하기 쉽게 고치면 아래와 같이 할 수 있다.

```
int function(int x, int y) {
    int result = 0;

    if (x <= 0) {
        return result;
    }
    result = x + y;
    if (y <= 0) {
        result += x * y;
    }
    return result;
}
```

두 코드의 차이점은 미미 하지만, 함수의 출구가 늘어남에 따라 코드의 조건문은 점점 평坦해 지는 것을 볼 수 있다. 실제 코드에서 만나는 조건문은 위와 같이 간단하지 않을 수도 있지만, 읽는 사람이 이해를 편하게 하기 위해서 평탄화 과정을 적용해 볼 필요는 있을 것이다. 중첩(Nesting)된 조건문은 코드의 가독성(Readability)에 악영향을 주기에 가능한 줄여야 하며, 평탄화 하는 방법이 그 해결책이 될 수 있다.

[파일(File)과 디렉토리(Directory)에 대해]

파일은 물리적으로 관련된 자료구조와 함수들을 묶어주는 역할을 한다. 파일들은 C언어에서는 사용하려는 각종 자료형을 정의한 헤더(Header) 파일과 자료형을 이용해서 원하는 기능을 구현하는 ".C" 구현 파일로 나누어진다. 물론, 그 외에도 각종 데이터 파일이나, 프로그램을 빌드(Build)하기 위한 파일들, 설정 파일들 등이 있을 수 있으나, 주로 다루어지는 파일은 이 두 가지 범주로 볼 수 있다.

이 두 가지 파일에 대해서는 "Copyright" 문구나, 파일의 관리자, 버전 번호들을 유지하기 위한 문구들이 주석으로 포함되기도 한다. 이런 파일들은 매번 유사한 주석을 추가하기 보다, 템플릿(Template)으로 만들어서 일관되게 사용하면 될 것이다. 또한, "Readme"나 "Install", "ReleaseNote"등과 같은 파일들을 만들어서, 배포나 설치에 유의해야 할 사항들을 추가해 된다.

파일도 이름이 중요하다. 즉, 파일에 내포된 기능들을 포괄해서 나타낼 수 있는 이름이 필요하다. 그 이름이 대표하는 것 이상의 일을 파일에 담아선 안된다. 바쁜 경우에는 파일의 이름만 가지고도 그 내부에 어떤 것들이 들어있는지 추측 할 수 있어야 한다. 요즘은 에디터(Editor)들이 좋은 것이 많아서, 쉽게 원하는 변수나 함수들을 찾을 수 있지만, 그래도 역시 파일은 좋은 이름을 가지는 것은 중요하다. 좋은 이름을 가진다는 것은 명확한 역할과 책임 범위를 정해서 만들었다는 것을 간접적으로 확인할 수 있기 때문이다.

파일의 이름을 주는 규칙은 팀에서 정하면 된다. 어떤 경우에는 파일의 이름 앞에 반드시 파일이 속한 모듈의 이름을 붙이기도 한다. 예를 들어 네트워크에 관련된 모듈에 정의된 파일이라면, "NET_"와 같은 모듈명을 붙여주기도 한다. 나머지 이름은 파일이 하는 역할을 묘사적으로 설명해주는 단어들이 나열될 것이다. "NET_SOCKET_CONNECT.c"와 같은 파일 이름을 가진다면, 이 파일에는 "socket"을 통한 연결에 대한 부분이 구현되어 있다고 생각할 수 있을 것이다.

파일의 이름을 대문자로 혹은 소문자로 줄 것인가는 알아서 정하면 된다. 주로 "Unix"와 같은 곳에서는 소문자를 주는 것이 일반적이다. 함수의 이름도 소문자를 "_"와 연결해서 사용한다. 명명 규칙은 어떻게 사용해도 되지만, 일관되게 사용하는 것이 전체적으로 깔끔하게 보이는데는 도움이 된다. 코드를 잘 관리하는 조직이라면, 이런 부분들이 깔끔하게 정리되었을 가능성이 높다. 코드를 열어보지 않더라도 이런 외관만보고도 대략적으로 코드의 관리 상태를 파악할 수 있다.

디렉토리도 마찬가지로 좋은 이름을 가져야 한다. 내부에 들어있는 파일들이 어떤 역할을 하는지 설명할 수 있어야 한다. 따라서, 디렉토리 이름은 대부분 모듈의 이름과 유사하다. 즉, 하나의 모듈을 하나의 디렉토리로 관리하는 것이 일반적이다. 디렉토리는 빌드의 단위도 되기에, 내부적으로 빌드 스크립트 (Script)를 가질 수 있다. 디렉토리 내부에 여러개의 모듈을 포함한 서브 디렉토리들을 가지고 있다면, 디렉토리 이름은 하나의 서브 시스템(Subsystem)과 같이 생각할 수 있다. 이 때도 마찬가지로, 모듈들을 대표하는 서브 시스템의 이름을 써주는 것이 좋다. 일반적으로 아키텍처 상에 그려지는 도형의 이름에 해당하는 디렉토리를 찾을 수 있으면 된다.

시스템은 여러 개의 서브 시스템으로 구성되며, 가장 상위 디렉토리에 그 시스템의 이름을 가진다. 내부에는 서브 시스템들의 디렉토리들로 구성될 것이다. 시스템에서 공통적으로 사용하는 "Include" 디렉토리의 경우에는 서브 시스템의 디렉토리와 분리될 수 있다. 하지만, 서브 시스템 내부에서 사용하는 "Include" 디렉토리는 서브 시스템 별로 생성해서 사용하는 것이 좋다. 가장 상위 디렉토리는 빌드를 시작하는 곳으로, 각각의 서브 시스템의 빌드를 시작하게 만들수 있다. 빌드는 서브 시스템 별로 진행하지만, 그 시작은 항상 최상위 시스템의 빌드서 시작하도록 만들어야 한다. 즉, 서브 시스템 별로도 빌드가 가능하지만, 전체적인 빌드도 한 번에 할 수 있도록 최상위 디렉토리에 만들어야 한다.

전체적으로 보았을 때, "시스템->서브 시스템->모듈(혹은, 컴포넌트)->파일->함수->변수" 등과 같이 점점 세분화 단계를 높여가면서 일관성을 유지하는 이름을 붙이는 것이 중요하다. 이것을 잘하고 있는 팀은 소스 코드에 대한 관리 원칙을 가지고 있으며, 항상 청결한 상태(Clean)로 코드를 유지할 가능성이 높다. 팀을 평가할 때 외부로 드러난 모습만으로도 간단히 팀의 상태를 파악할 수 있는 잣대로 사용할 수 있다. 물론, 그렇다고 팀의 역량이 완전히 그런 것으로 평가된다는 것은 아니다. 간단히 어떤 상태인지를 아는 정도에서 그치는 것이 좋다. 어쨌든 체계적인 "이름주기 규칙(Naming Rule)"을 정하는 것은 코드에 대한 이해를 높이는데도 도움이 된다. 어떤 구조로 코드가 구성되었는지에 대한 단서를 제공하기 때문이다.

[코드의 복잡성(Complexity) 문제를 해결하는 방법]

소스 코드가 복잡해지는 이유는 코드를 잘못 짜기 때문이다. 하지만, 대부분의 개발자들은 자신의 코딩 스타일에 대해서는 문제삼는 것을 거부한다. 코딩에 문제가 있다는 것을 코드를 작성하는 사람의 역량이 미흡하다는 말과 동일시 하는 경향이 있다는 뜻이다. 사람의 성향상 자신의 노력과 시간이 들어간 부분에 대해서는 다른 사람이 생각하는 가치보다 높게 보는 것이 일반적이지만, 코드에 대해서는 적어도 "자신의 것"이라는 생각을 버리는 것이 현명하다.

코드의 소유권(Ownership)에 대한 문제는 개인 차원에서 생각해서는 안된다. 팀에서 작성된 모든 코드는 공동의 소유이며, 회사에서 작성한 코드는 회사의 소유다. 자신의 코드를 남에게 보여주지 않거나 코드 리뷰의 검토 대상에서 제외하려고 생각한다면, 잘못된 생각이니 바꾸기 바란다. 자신이 작성한 코드라도 자신의 것이 아니며, 모든 코드는 공유되어야 할 것들의 일부일 뿐이다.

이런 관점에서 복잡성을 해결하기 위해서는 적극적인 코드 리뷰를 실시해야 한다. 그리고, 코드 리뷰의 핵심은 "가독성(Readability)"이다. 코드가 쉽게 읽힐 수 있도록 만들어야 리뷰가 가능하고, 기능의 추가나 변경이 용이하다. 따라서, 읽기 쉬운 코드의 특징을 이해하고, 이를 코딩에 적극적으로 적용하는 것이 핵심이 되는 것이다. 만약, 어떤 코드를 읽었을 때 이해하기가 어렵다면, 그 코드는 다시 작성할 필요가 있다. 자신이 만든 코드를 읽는 것은 쉽지만(시간이 흐르면 이것도 힘들 수 있지만), 다른 사람의 코드에 대한 거부감이 생기는 이유도 마찬가지다.

- 01. 의미있는 이름을 사용한다.**
- 02. 짧은 함수를 많이 만들어서 사용한다.**
- 03. 함수의 인수는 가능한 적어야 한다.**
- 04. 함수의 복귀 시점은 가능한 한 곳으로 유지하지만, 이유가 명확한 경우에는 여러 곳이 될 수 있다.**
- 05. 조건문은 가능한 적은 조건을 검사하고, 가능한 긍정적인 조건문을 사용한다.**
- 06. 자료구조에 대한 직접적인 접근을 막는다.**
- 07. 전역 변수는 사용하지 않는다.**
- 08. 초기화는 모아서 한번에 할 수 있도록 해준다.**
- 09. 할당된 메모리는 동등한 수준에서(함수, 모듈등) 반드시 해제가 되어야 한다.**
- 10. 가능한 예외상황을 가정해서 방어적으로 코딩 한다.**
- 11. 테스트를 만들어서 함수 단위로 실행해서 결과를 알 수 있어야 한다.**
- 12. 복제된 코드는 제거한다.**
- 13. 표준 코딩 스타일을 사용하도록 한다.**
- 14. 코딩 룰을 검사할 수 있는 도구를 사용한다.**
- 15. 정적분석 도구를 사용한다.**
- 16. 작성된 코드는 다른 사람의 코드와 통합되어(Merge) 빌드(Build)가 되어야 한다.**
- 17. 자동화된 테스트를 가능한 빨리 실행해야 한다.**
- 18. 자동화된 배포가 가능한 수준까지 필요한 시스템을 지속적으로 구축한다.**
- 19. 컴파일러의 경고 수준을 가능한 높이고 경고 메시지는 전부 제거한다.**
- 20. 계층화를 지향하며, 계층을 위반하는 호출을 제거한다.**

너무 많다고 생각한다면, 차근차근 하나씩 해보기 바란다. 이런 것들을 한번에 다 하는 것은 힘들지만 지속적으로 조금씩 해보는 것은 꼭 필요하다. 제대로 일하기 위해서는 기초를 충실히 닦아야 하며, 위에서 나열한 것들은 기초로 삼아도 좋은 것들이다. 일하는 동안 배울 수 있는 기회가 같이 따라온다면 좋겠지만, 실제 업무에서는 "아무도 그렇게 안한다"는 말을 들을지도 모른다. 하지만, 그 "아무도"에는 끼지 말고 꼭 해보기를 바란다.

소프트웨어는 복잡성을 관리하는 법을 알아야 제대로 만들 수 있다. 대부분의 소프트웨어가 한계에 부딪히는 부분이 바로 이런 것들이며, 지속적인 성장을 가능하게 만들려면 코드가 지나치게 어려워지는 것을 막을 수 있는 방법들을 처음부터 만들고 시작해야 한다. 그렇지 않으면 얼마 지나지 않아 힘든 상황이 오게되며, 몇 번 우연적으로 성공했다고 하더라도, 결국에는 그 성공에 발목이 잡힐 수 있다. 그렇게 하지 않으려면 차근차근 하나씩 성실하게 과정을 밟아 나가야 한다.

[코드 리뷰 가이드 Version 1.0]

소프트웨어 개발에서 가장 많은 버그를 찾아낼 수 있는 것은 "정형적인 코드 리뷰"라고 알려져 있다. 하지만, 실무에서는 가장 잘 지켜지지 않는 부분이기도 하다. 형식을 갖추고 진행하는 코드 리뷰는 미리 리뷰할 코드를 정하고, 참석자를 선정하게 되며, 정해진 과정을 통해서 코드를 읽고 설명하는 과정을 통해서 리뷰가 진행된다. 차선책으로 선택할 수 있는 방법은 "비 정형적인 코드 리뷰"이며, 주로 "온라인 코드 리뷰"나 "페어(Pair) 프로그래밍"과 같은 코딩과 병행하는 코드 리뷰 등이 있다. 또는, "책상에서 개인이 출력된 코드를 가지고 하나씩 검증하는 방법"도 있다. 이런 부분들이 잘 실행되지 않는 이유는 코드 리뷰에 대한 부정적인 생각들이 많다는 점이다. "코드 리뷰를 해봐야 잡을 수 있는 버그는 별로 없다", 혹은 "

과제가 일정이 코앞인데, 코드 리뷰할 시간이 없다"라는 생각들이 주를 이룬다. 사실 이것도 일정 부분은 사람의 심리에 깊숙히 뿌리박힌 눈앞의 손실을 회피하는 것(혹은, 단기적인 이익을 추구하는 것)에 그 이유를 찾을 수 있을 것이다. 그리고, 더 중요한 점은 코드 리뷰가 불편한 자리라는 점이다. 즉, 자신이 작성한 코드를 남에게 보여주고 비평을 듣는 자리이기 때문이다. 어떤 식으로든 그런 것들을 피하는 것이 일단은 자신을 방어하는 최선의 방법이기 때문이다. 하지만, 결국 이런 "불안함과 불편함"을 극복할 수 없다면, 나중에 오는 길고 지루한 버그 찾기의 과정은 어쩔 수 없는 결과로 받아들여야 한다.

코드 리뷰를 위해선 먼저 기준이 필요하다. 코드를 얼마나 리뷰할지, 얼마동안 리뷰할지, 어떤 코드를 대상으로 할 것인지 등을 정해야 한다. 그리고, 리뷰를 위해서 누가 코드를 볼 것인지도 중요하다. 리뷰의 기준도 필요하며, 이를 코드 리뷰를 하는 사람들이 정확히 찾아서 이야기 해주어야 하기 때문이다. 마지막으로 코드 리뷰를 했던 결과물을 어디에 정리할 것인지와 개선 사항을 확인하는 과정도 포함되어야 한다. 이런 것들이 일종의 코드 리뷰에 대한 상세 프로세스를 이루며, 그 과정의 내용(Contents)들은 조금씩 자신의 팀에 맞게 다르게 만들 수 있다. 아래는 소프트웨어의 코드 품질 측면에서 특히 중요하게 다루어야 할 10가지만 정리한 것이다. 코드 리뷰의 체크 리스트 정도로 생각하면 될 것이다.

01. 코드에는 "마법의 수(Magic Number)"와 같은 아무런 설명이 없는 상수 값의 사용이 없어야 한다.

; 마법의 수는 항상 막성을 일으킨다. "왜"라는 질문에 대답해 줄 수 있는 근거를 코드에서 찾을 수 없기 때문이다. 이런 것들이 많은 코드는 고치기가 당연히 어렵다. 왜라는 질문을 하지 않을 수 있는 코드를 만드는 것이 "읽기 좋은 코드"를 만드는 핵심이다.

02. 주석 처리된 코드는 없어야 한다. 있다면 반드시 삭제해야 한다. 또한, 필요없는 주석도 제거해야 할 것이다.

; 주석은 순기능과 역기능이 있다. 어떤 사람들은 주석이 얼마나 코드 내에 분포하는 가를 가지고 코의 품질을 측정하는 경우도 있다. 하지만, 기본적으로 주석이 없어도 이해가 되는 코드를 만드는 것을 목적으로 해야한다. 그리고, 당연히 주석으로 존재하는 코드는 없는 것이 코드를 읽는 사람의 혼동을 줄여줄 것이다. 그런 코드들은 지우기도 애매하고, 유지하면 다른 코드를 읽는데 방해가 될 뿐이다. 차라리, 스스 코드 관리 시스템을 이용해서 흔적을 남겨놓는 것이 좋다.

03. 전역 변수는 될 수 있으면 없어야 한다. 어쩔 수 없이 사용하는 경우도 있지만, 한정된 범위에서만 사용된다는 것을 확인해야 한다.

; 전역 변수는 의존성에 치명적인 영향이 있다. 가능한 사용되는 범위를 줄이는 것이 최고의 해결책이다. 아예 사용하지 않으면 더 좋겠지만, 사용해야 하는 경우도 종종 있다. 어쨌든 전역 변수를 사용하는 것들 간에는 서로 "의존성(Dependency)"을 만들어, 코드 변경의 영향을 주고 받을 수 밖에 없다. 소프트웨어 개발의 핵심은 "의존성을 줄이고, 응집성을 높이는 것"이 핵심이다.

04. 모든 변수, 함수, 파일, 디렉토리 등은 적절한 이름을 가져야 하며, 사용되는 역할과 목적을 의미하는 정보를 제공해야 한다.

; 사람의 이름과 마찬가지로 코드에서도 이름이 중요하다. 소설에 등장 하는 이름이나, 영화 속의 주인공들은 전부 자신의 역할이나 성격에 맞는 이름을 사용한다는 것을 유의해서 봐야 할 것이다. 마찬가지로 코드에서도 변수나 함수 등등 다양한 것들이 자신의 역할이나 성격에 맞는 적절한 이름(배역)을 가져야 한다. 그것을 통해서 코드를 읽는 사람은 상세한 것을 몰라도 무슨 일을 하는지(무엇에 사용되는지)를 추측할 수 있게 되기 때문이다.

05. 변수는 적절한 타입으로 선언되어야 하며, 일관되게 사용해야 한다. 타입의 캐스팅을 가능한 줄이도록 해야 한다.

; 변수의 타입은 변환에서 자유롭지 못하다. 모든 변수가 가질 수 있는 값의 범위나 변화 정책에 대해서 일일이 기억하기는 힘들다. 물론, 컴파일러가 적절한 경고 메시지를 출력해 주는 경우도 있지만, 될 수 있으면 일관된 타입을 사용하는 것이 좋다. 타입을 변환(Casting)해서 다른 타입으로 사용하는 것은 가능한 줄여야 한다. 그것으로 인해서 오차가 발생할 가능성이 있거나, 의도하지 않은 결과를 만들 수 있기 때문이다. 특히, 부동 소수점과 같은 타입을 사용할 때는 같은 타입으로 사용하는 것이 좋고, 될 수 있으면 조건문에서는 사용하지 않는 것이 바람직하다.

06. 모든 조건부 문들은 제대로 실행될 수 있는지 확인되어야 한다. 또한, 가능한 "switch()"문과 같이 여러 분기를 하는 코드는 줄여야 하며, 만약, 사용할 경우에는 반드시 각각의 "case"에는 "break"를 붙여야 하고, "default"를 가져야 한다.

; 조건 분기문은 코드를 이해하기 어렵게 만드는 "복잡도(Complexity)"를 증가시킨다. 그리고, 어떤 조건문의 경우에는 절대 참이나 거짓이 실행되지 않는 경우도 있다. 이런 부분들을 사람이 일일이 판단하기는 어렵기에 대부분 도구나 자동화된 테스트의 커버리지를 가지고 점검하게 된다. 어쨌든, 가능한 적은 분기문을 가지는 것이 코드를 이해하는데 도움이 되며, 만약 사용해야 할 경우에는 될 수 있으면 확실히 모든 요소를 다 가지도록 만들어야 할 것이다(switch(), case, break, default). 간혹 실제로 사용되지 않는 분기를 가지고 있는 경우가 있으며, 이와 같은 분기문들은 삭제하는 것이 현명하다(잘 몰라서 남겨두는 경우가 많다.).

07. 조건 분기문은 가능한 2 수준(Level)의 중첩(Nesting)으로 한정한다. 조건은 부정문 형태보다는 긍정문 형태로 사용하는 것을 선호한다.

; 조건이 중첩되어 있으면 코드를 이해하기는 더 힘들어 진다. 따라서, 여러 번의 중첩이 발생하는 경우에는 따로 떼어내는 것이 코드를 이해하는데 도움이 된다. 부정보다는 긍정적인 조건문이 이해하기 쉬우며, 될 수 있으면 대부분의 경우가 미리 걸러질 수 있도록 조건문을 배치해야 할 것이다(더 많이 실행될 가능성이 있는 조건). 2단계 이상의 조건문의 중첩을 허용하지 않는 것이 바람직하며, 테스트도 조건이 간단해 질수록 쉬워지는 경향이 있다. 그것이 어렵다면 일부 조건문이나 하위 실행에 관련된 문장들을 따로 분리해서 함수로 만들 수 있다. 조건문을 풀어서 평탄하게 만드는 것(flat)도 한가지 방법이다.

08. 함수의 길이는 100라인 이하로 권장한다(작으면 작을 수록 좋다). 함수의 파라미터 최대 개수는 4개 이하로 한정한다.

; 함수는 절대 여러가지 일을 해선 안된다. 정의된 한가지 일을 해야한다. 그리고, 함수 내에는 그 함수의 이름보다 한 수준 정도 아래의 추상화를 가지는 문장들로 구성하는 것이 좋다. 그 문장들은 다른 함수가 될 수도 있으며, 혹은 직접적인 상세 구현이 될 수도 있다. 대부분 함수는 짧을 수록 이해하기도 더 쉽다. 따라서, 가능한 짧게 함수를 만드는 것이 좋다. 물론 언어에 따라 함수의 길이는 달라질 수 있기에, java나 C#과 같은 경에는 20라인보다 더 짧은 기준을, C나 C++의 경우는 50라인 이하의 기준을 가져가는 것이 좋을 것이다. 그보다 길어지면 새로운 함수를 만들어서 분리하거나, 여러 개의 함수로 쪼개는 과정이 필요하다. 이때 함수의 깊이가 너무 깊지 않도록 유지하기 위해서는 각각의 함수가 "호출->복귀"의 연속이 되도록 추가적인 상위 함수를 정의할 수 있다. 함수가 하는 일이 줄어드면 함수에서 처리해야 할 데 이터 종류도 줄어들게 되며, 만약 종류를 줄이지 못할 경우에는 한 덩어리의 데이터(자료구조)로 묶어서 관리할 수도 있다.

09. 함수가 리턴 값(Return Value)이 있을 경우에는 반드시 검사해야 한다. 오류가 발생할 수 있는 상황이라면, 오류에 대한 일관된 처리를 하고 있는지 점검해야 한다. 함수의 입력 값에 대해서도 유효한 범위를 가지는지 확인하도록 한다.

; 오류는 항상 발생할 수 있다. 시스템에서 제공하는 파일 오픈이나 메모리 할당, 소켓의 생성 등 대부분의 경우에는 오류가 발생하지 않는 함수 들도 특정 상황에서는 오류가 발생한다. 따라서, 그런 것들을 대비하기 위해서는 방어적인 코딩을 해야한다. 따라서, 모든 리턴 값이 있는 함수에는 방어적으로 복귀값을 확인하는 것이 기본이다. 자신이 만든 함수건 누가 만들어준 함수건 상관없이 방어적으로 코드를 만들어야 한다. 만약, 함수가 특정 범위의 입력 값 만을 요구할 경우에는 넘겨주는 값이 그 범위에 있는지도 확인해야 한다. 물론, 이로 인해서 발생하는 오버헤드는 있겠지만, 결과적으로 시스템은 더 안정적으로 동작하게 된다. 만약, 특정 입력 값 만이 전달될 것으로 가정할 수 있다면, 디버그 시에만 방어적인 코드가 동작하도록 컴파일 옵션으로 처리할 수도 있을 것이다. 함수가 돌려주는 값에 따라 일관된 처리 방법을 제공하면, 그것을 사용하는데 있어서 불편을 최소화 시킬 수 있으며, 프로그램의 오류 정보에 대해서도 더 많은 것을 관리할 수 있을 것이다.

10. 각각의 파일들은 지나치게 길지 않은지 확인해야 한다. 파일의 길이는 대략 1,000라인 이하를 가지도록 만들어야 한다. 지나치다는 말은 7+9개 이상의 변수, 함수, 내부 변수 등등이 있다는 말을 기준으로 할 수 있으나, 내부 함수의 경우에는 관련성이 명확히 없거나, 지나치게 다른 모듈에 의존적인 부분들이 있는지 점검해야 한다.

; 사람이 한번에 기억할 수 용량은 한계를 가진다. 특히, 단기기억 저장소 같은 경우에는 대부분 7+2 정도의 개수를 위해서 사용한다. 코드를 이해하기 쉽게 만들려면 너무 많은 기억할 것들을 만들지 않아야 한다. 그리고, 추상적인 사고를 돋기 위해서는 상세화 수준에서 항상 7+2를 유지하는 것이 도움이 될 것이다. 같은 수준의 추상화를 가지는 것들은 항상 5개에서 9개 수준으로 기억되도록 만들어, 한꺼번에 이해할 수 있도록 코딩 한다. 그 보다 많은 수가 있다면, 일단은 코드를 더 잘게 나눌 수 있는지 고민해야 할 것이다. 그렇지 않다면, 묶어 줄 수 있는 것들을 최대한 묶어서 추상화 시키는 것이 좋다. 만약, 함수나 파일, 혹은 모듈이 맡은 역할에 속하지 않는다고 판단된다면, 과감하게 다른 곳으로 옮기거나 새로운 함수, 파일, 모듈을 만들어서 분리하는 것도 한 가지 방법이다. 가능한 작고(적고) 짧게 유지하려고 노력해야 한다.

위에서 나열한 것들은 코드 리뷰를 위한 체크 리스트 정도로 활용하면 될 것이다. 필요한 경우 더 추가할 수 있다. 우선 순위는 중요하지 않지만, 만약 팀에서 이미 암묵적으로 동의하는 부분에 대해서는 체크 리스트를 만들지 않아도 되며, 가장 중요하다고 생각하는 것 몇 가지만 추려서 사용할 수도 있다. 유지보수 측면에서 중요한 점들을 나열한 것 이기에, 팀마다 다른 체크 리스트를 가질 수 있으며, 자신의 팀에 맞는 것으로 적절히 편집해서 사용하면 된다.

코드 리뷰는 사람이 하는 개발 활동 이기에 "반드시 해야하는 일"로 취급해야 한다. 이 점은 사실 중요한 부분이다. 대부분의 관리자나 개발자가 코드 리뷰를 해야하는 일로 보지 않고, 부가적인 업무 정도로 생각하기 때문이다. 즉, 코드 리뷰가 귀찮은 일이며 될 수 있으면 안하는 것이 좋다고 생각한다. 물론, 이것은 잘못된 생각이다. 따라서, 업무 스케줄을 정할 때 반드시 코드 리뷰에 대한 시간을 고려해서 계획을 짜는 것이 좋다. 그리고, 관리자는 그것이 "해야하는 일"로 생각하도록 구성원들을 이끌어야 한다. 코드 리뷰를 일상적으로 하는 것이 부담스럽다고 생각한다면, 하루의 일과 중에 일정 시간을 코드 리뷰만 하는 시간으로 예약해두는 것도 좋을 것이다.



예를 들어, 오전 10시부터 12시까지는 어떤 회의도 참석하지 않고, 지시도 미뤄두고 코드 리뷰를 하는 시간을 갖는 것이다. 너무 오래 동안 리뷰를 하는 것은 문제점을 찾는데 도움이 되지 않기에, 2시간 이상을 하는 것은 생산성에 도움이 되지 않는다. 그리고, 한번에 400라인 이상도 좋지 않다. 따라서, 대략 50분정도 200라인을 상세히 보고, 10분 휴식 후에 다시 200라인 정도 보는 것이 좋을 것이다. 코드 리뷰를 한번에 몰아서 하는 방법은 좋지 않다. 전체가 모여서 코드를 리뷰 해야 할 경우는 반드시 팀원들이 이해해야 한다고 보는 "중요 코드"에 한정하는 것이 좋고, 일상적인 온라인 리뷰의 경우에는 모든 변경된 코드에 대해서 자주 리뷰하는 것이 좋다. 이를 위해서는 개발자들은 자주 저장소의 코드와 자신의 변경을 통합하는 것이 유리할 것이다. 통합은 늦어질수록 항상 비용이 많이 든다는 점을 기억해야 한다.

코드 리뷰는 자신이 옷을 벗고 팀원들의 앞에 나서 누드 모델이 되는 것이 아니다. 따라서, 방어적인 자세보다는 오히려 적극적으로 의견을 받아들이는 것이 더 좋다. 코드 리뷰의 부가적이 장점은 사람들이 가지고 있는 지식을 서로 나눌 수 있다는 점이다. 코딩에 대한 지식과 도메인(Domain)의 이해를 돋는 지식 등 다양한 견해들을 나눌 수 있다. 물론, 그렇다고 모든 의견에 대해서 전부 수정하는 것이 옳다는 것은 아니다. 정당한 이유와 설명이 있다면 자신의 코드를 변경하지 않을 수 있다. 중요한 점은 "코드와 자신의 인격을 동일시 하지 말아야 한다"는 것이다.

흔히 사람들이 불안과 공포를 느끼는 것은 자신의 "자식과도 같은" 코드를 남들이 비판하는 자리라고 여기기 때문이다. 이는 코드의 소유권에 대한 문제를 내포하는 것으로, 사실 코드가 자신의 손을 떠나면 자신의 것이 아닌 "모두의 것"이 된다고 봐야 한다. 작은 소규모 기업의 개발자들은 흔히 자신이 만든 코드를 "자신의 것"이라고 생각하는 경우가 많지만, 엄밀히 말하면 회사에서 돈을 주고 개발한 코드는 전부 회사의 소유가 되는 것이 맞다(법적으로도). 물론, 집에서 혼자 만든 코드는 자신의 것으로 해도 무방할 것이다. 하지만, 회사에서 얻는 특별한 기술을 사용해서 회사 제품과 경쟁하는 제품을 만드는 것은 도의상 옳지 않다. 이는 개발자의 기본 자질에 대한 것 이기에, 스스로 항상 조심스럽게 생각해야 할 부분이다.

코드 리뷰는 남을 돋는 과정이지 남에게 맞서는 것이 아니며, 결과적으로 상향 평준화를 이루는 길이지 경쟁을 하는 자리가 아니다. 경쟁은 누가 더 멋진 코드를 만들어 내는가에 달려있는 것이지, 남의 코드를 비난하고 자신의 코드가 우수하다고 주장하는 것이 아니다. 리뷰는 어떤 일이 제대로 되었는지를 확인하는, 사람이 할 수 있는 타인을 위한 아름다운 선행이다.

[코드 리뷰 가이드 Version 2.0]

기본적인 코드 리뷰를 잘하고 있는 조직이라면, 기초적인 것들을 챙기기 보다는 조금 더 수준을 높일 필요가 있다. 여기서는 그런 사람들을 위해서 도움이 될 만한 것들을 나열해 보기로 하겠다. 물론, 여기서 제시하는 것들은 전적으로 개인의 생각이며, 어떤 목적을 가진 것은 아니다. 나열 순서에 대해서도 우선 순위는 없으며, 그냥 참고로 사용할 수 있을 정도의 수준일 뿐이다. 동의하지 않는 부분이 있을 수도 있으니, 도움이 될 만한 것들만 추려서 사용하면 될 것이다.

01. 함수 길이가 30라인 이하인지 확인한다.

; 어떤 언어를 사용하더라도 코드의 길이는 짧은 것이 좋다. C/C++, Java, C#등의 언어와 상관없이 될 수 있으면 최소 실행단위를 짧게 만들어야 한다. 이해하기 쉽고, 테스트 하기 쉽고, 재활용하기 쉽고, 수정하기도 더 쉽기 때문이다. 유지보수에서 핵심적인 것은 코드의 길이이며, 반드시 관리해야 할 항목 중에 하나다.

[참고] Google의 C/C++ 코딩룰에서는 함수의 길이를 50라인을 추천한다. Linux의 경우에는 24라인을 사용하라고 이야기하고 있다. 이는 가장 열악한 개발 환경에서도 한 화면에 볼 수 있는 코드의 길이를 나타낸다.

02. 단위 테스트가 코드를 95%이상 검증 하는지 확인한다.

; 단위 테스트는 선택이 아니라 필수다. 물론, 실무에서는 이런 부분들이 간과되거나, 시간이 없다는 말로 무시 되기도 한다. 하지만, 조금 더 나은 수준의 개발자가 되기 위해서는 한번은 넘어야 할 산이다. 습관화시키는 것이 중요하며, 될 수 있으면 한번에 많이하지 말고, 자주 작게 하는 것이 바람직하다.

03. 주석이 코드와 일치하는지 확인한다.

; 주석이 없이도 이해가 되는 코드를 만드는 것이 중요하지만 주석이 필요할 때도 있다. 주석과 코드는 동기화가 맞지 않는다는 문제점이 있기에, 이를 코드 리뷰에서 반드시 확인할 수 있어야 한다. 주석은 코드를 해석하는 것이 아니라, 그렇게 만든 이유를 설명하는 것이다. 그리고, 특별한 양식으로 코드를 다큐먼트로 만들 경우에도 사용할 수 있다. 하지만, 너무 과하게 사용하지는 말아야 한다. “과한 것은 부족한 것보다 못한 법”이다.

04. 자료구조(내부 구현 구조)의 노출이 없는지 확인한다.

; 내부 구현 정보를 될 수 있으면 외부에서 알지 못하게 하는 것이 좋다. 그런 정보의 노출이 늘어날수록 코드 간에는 의존성이 생기기 마련이다. 인터페이스만 공개하는 것이 최선이며, 상속(Inheritance)과 같은 부분도 될 수 있으면 줄이는 것이 좋다. 상속도 결국 그 상속의 고리와 상속으로 인한 의존성을 만들어 내기 때문이다. 물론, 상속이 객체지향 프로그래밍에 중요한 요소이지만, 상속보다는 인터페이스를 선호한다.

05. 코드가 아키텍처와 일치하는지 확인한다.

; 아키텍쳐 문서를 만드는 것은 전체적인 시스템의 공통된 뷰(View)를 개발자가 공유하기 위해서다. 그리고, 시스템을 설계한 목적과 결정의 근거를 남기는 역할도 있다. 하지만, 구현에 들어가면 생각했던 것과는 다른 부분이 존재하기 마련이며, 이때 아키텍쳐를 기술한 문서와 구현된 코드의 불일치가 발생한다. 이것은 문서가 관리되고 있지 않다는 것을 의미한다. 따라서, 둘 간에 정합성이 있는지 확인해야 한다.

06. 리팩토링(Refactoring)이 필요한 코드나 테스트는 없는지 확인한다.

; 냄새가 나는(Bad Smell) 코드를 발견한다면 될 수 있으면 수정하는 것이 좋다. 테스트를 만들었다면, 테스트 역시 코드의 일부로 취급해서 관리해야 한다. 테스트도 마찬가지로 중복을 없애는 것이 좋으며, 코드는 당연히 깨끗한(Clean)한 상태로 만들어야 한다. 여러가지 이유로 냄새나는 코드가 스며들 가능성이 많기에, 냄새를 막는 방법도 익히고 그것을 해결하는 방법도 알아야 할 것이다.

07. 최적화된 빌드(Build)에서도 테스트가 제대로 동작 했는지 확인한다.

; 나중에 가서 빌드(Build)를 최적화하는 것은 언제 터질지도 모르는 시한 폭탄을 안고가는 것과 같다. 물론, 그렇다고 반드시 터진다는 보장은 없다. 하지만, 이를 미연에 방지하기 위해서 항상 배포할 수 있는 수준으로 코드를 유지하는 것이 필요하다. 따라서, 최적화된 빌드에서도 문제가 없는지 확인해야 한다. 그렇다고 최적화를 위해서 코드를 만들라는 것은 아니다. 코드의 최적화는 항상 구조화 다음이다.

08. 특정 코딩 표준 툴(Tool)의 검증을 통과 했는지 확인한다.

; 팀에 코딩 표준을 가지더라도 버그의 유발 가능성은 될 수 있으면 줄이는 것이 좋다. 정적 분석이나, 코딩 표준 검사 툴 등을 이때 활용할 수 있을 것이다. 문제는 빨리 찾아서 해결하는 것이 비용을 줄이는 길이며, 컴파일러의 경고 메시지들과 같은 것도 될 수 있으면 제거해야 한다. 그대로 두면 나중에도 그대로 남는다. 그것이 아무리 사소해도 문제는 문제인 것이며, 항상 청결하게 코드를 유지해야 한다.

09. 추상화가 일관되게 적용되는지 확인한다.

; 구체적인 것과 추상적인 것을 섞어서 코딩하면, 코드는 이해하기 어렵게 된다. 그리고, 계층화를 위반할 가능성도 높다. 추상화는 생각을 표현할 수 있는 방법이며, 구체적인 내용을 모르고도 코드를 편하게 읽을 수 있도록 만든다. 따라서, 코드의 최소 실행 단위인 함수와 같은 곳에서 일관된 추상화 수준을 맞

줘 주어야 할 것이다. 어긋나는 부분을 끊어서 새로운 함수로 추상화 시킬 수 있다.

10. 의존성이 구현과 분리 되었는지 확인한다.

; 상위 계층에서 사용할 인터페이스와 하위 계층에 대한 의존적인 인터페이스를 분리하는 것은 모듈의 재 사용성을 높여줌과 동시에, 구현을 정의와 분리시켜 코드의 수정을 쉽게 만든다. 의존성을 관리하는 것이 소프트웨어 개발의 핵심이며, 상세 구현에 대한 것은 인터페이스 너머로 감춰 주어야 한다. 구체성이 외부로 노출되면 의존성은 필수적으로 따라오게 된다.

이상과 같이 열 가지를 살펴보았지만, 사실 이런 부분들을 일일이 확인하는 것은 쉬운 일이 아니다. 일부는 툴로 대체를 할 수 있고, 일부는 사람이 직접 코드를 읽어야 확인할 수 있는 부분이다. 따라서, 가능한 자동화 할 수 있도록 개발 환경을 구축하는 것이 좋다. 예를 들어, 항목 "01, 02, 05, 07, 08"의 경우에는 자동화가 가능한 부분이다. 나머지 부분은 시간이 조금 걸리겠지만, 사람이 직접 확인해야 할 것 들이다.

이런 가이드가 필요한 이유는 다음 단계로 "무엇을 해야할지"를 결정하지 못했을 때다. 기본적으로 생각할 수 있는 것들을 정의하고 나면, 일단은 생각의 기준(Anchor)이 마련되는 것이다. 이 기준은 중요하다. 사람들은 자신이 생각하는 것의 기준이 될 만한 것들을 찾지 못하는 경우가 많으며, 남이 만든 기준이라고 하더라도, 한번 정하게 되면 그것을 통해서 새로운 자신 만의 기준을 만들 수 있기 때문이다. 목표를 약간 높이정하거나, 혹은 가격을 흥정할 때 낮춰서 잡는 것도 이런 효과를 보기 위함이다.

상위 수준의 가이드 라인을 적용하기 위해서는 팀원들이 동의하는 것들을 찾아야 한다. 당연히 "왜"라는 질문에 대해서 논리적이고 이성적인 답이 필요하다. 하지만, 항상 그렇듯이 "논리적이고 이성적인 답"은 아무런 행동 변화를 일으키지 못한다. 따라서, 가이드 라인을 적용할 때는 신중한 접근이 필요하다. 한가지 방법은 좋은 질문을 통해서 하는 것이다. "우리는 오늘 우리가 만드는 소프트웨어의 품질을 높이기 위해서 최선을 다했나요?"와 같은 것이 사람들의 감정 변화 및 주체 의식에 대한 의지를 상기시켜 줄 수 있을 것이다.

우린 일을 하면서 배운다. 하지만, 배움이란 배우고자 하는 사람에게만 혜택을 준다. 일을 하면서도 배우지 않는 사람은 "일의 의미"를 찾지 못하는 사람들이다. 사실 지금하고 있는 일이 자신의 천직이라고 생각하는 사람들은 별로 안될 것이다. 하지만, 10년 이상을 같은 분야를 전공한 사람들은 자신이 지금하고 있는 일이 "자신이 정말 찾고 싶었던 일"인 경우를 만나게 된다. 관심이 없던 일이라고 하더라도, 꾸준히 노력하면 관심이 생길 수 있다. 그래도 별로 발전도 없고 관심이 생기지 않는다면, 일을 포기하기보다는 바라보는 자세를 달리하는 것이 필요하다. 하나에 집중하는 것은 충분히 했으니, 그 주위의 것들로 넓혀 나갈 때란 것이다. 이런 과정을 통해서 기존의 시각과 사물을 다르게 보는 방법을 훈련할 수 있으며, 관심의 폭 또한 점차 커질 것이다.

- 수준을 상향 평준화 하라. 도전적인 목표를 제시하고, 다가가는 방법을 함께 고민하라.-

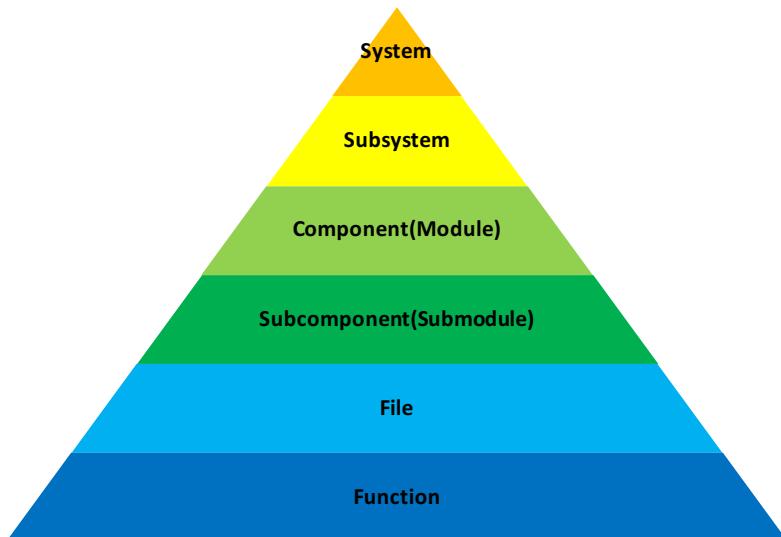
4. 프로그램의 구조

프로그램의 구조는 구현하는 조직의 구조를 반영하거나, 혹은 정의된 프로그램 구조로 조직의 구조를 변경하기도 한다. 동일한 일은 하는 프로그램이라고 하더라도 구조는 달라질 수 있으며, 이러한 “달라짐”的 이유는 “사람이 하는 일”이기 때문이다. 즉, 사람은 자신이 하고 싶은 일을 어떻게 할지 자유로운 선택을 할 수 있으며, 그것에 따라 프로그램의 구조도 달라 지게 된다. 문제는 프로그램의 구조가 구현 및 유지보수에 영향을 주게 된다는 점이다. 즉, 어떤 구조를 가지느냐에 따라 “비용(Cost)”이 달라질 수 있다. 따라서, 이번 장에서는 프로그램이 어떤 구조(물리적, 혹은 논리적)를 가지는 것이 좋은지 보도록 하겠다.

[모듈화란?]

모듈이란 독립적으로 재활용될 수 있는 소프트웨어 덩어리를 말한다. 하나의 모듈은 자신을 사용하기 위한 API를 제공하고 있으며, 자신이 포팅되기 위해서 필요한 명확한 환경을 나열할 수 있어야 한다. 모듈화를 한다는 말은 잘 정의된 인터페이스 이외에는 자신의 내부구조에 대한 직접적인 접근을 막는다는 의미이며, 다른 환경으로의 포팅도 자신이 사용하는 특정 인터페이스에만 의존적이란 의미다. 모듈은 독립적으로 테스트 될 수 있으며, 다른 모듈에 대한 의존성을 최소한만 가져야 한다. 모듈화는 제공되는 인터페이스를 지원하는 어떤 모듈로도 대체되는 것도 가능하다.

모듈은 계층적인 소프트웨어 구조를 설계할 때, 한 계층의 부분을 차지하는 단위가 된다. 즉, 계층화를 통해 제공해야 할 인터페이스와 자신이 의존하고 있는 인터페이스를 명확히 분리해 내야한다. 하나의 모듈은 상위 계층에 제공하는 인터페이스, 실제 자신이 담당하고 있는 역할을 처리하는 부분, 제공되는 인터페이스를 사용하기 위한 의존성을 가지는 부분등 3개의 작은 계층으로 나누어질 수 있으며, 이렇게 만들어진 모듈은 이식성(Portability)과 호환성(Compatibility)이 높다. 분리된 계층을 가지지 못한 모듈들은 내부 구조의 변경이 상위 모듈에 영향을 줄 수 있으며, 이런 형식의 구현은 구체적인 부분을 외부로 노출하게 된다.



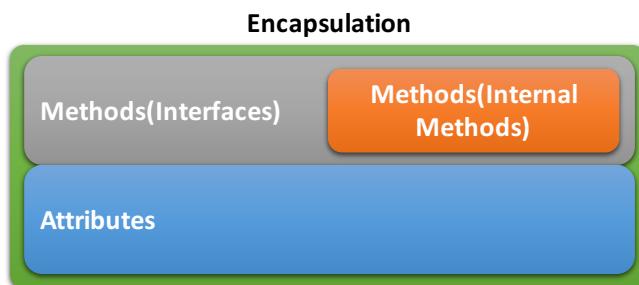
모듈도 함수와 마찬가지로 하나의 역할만 해야한다. 즉, 하나의 책임을 가지고 있다. 그 책임의 폭이 달라질지는 몰라도, 정확히 정의되는 하나의 역할을 맡고 있어야 하며, 그것 이외에 다른 일을 해서는 안된다. 사실 이 부분이 가장 힘들다. 대부분의 경우 지나치게 많은 역할을 정의해서, 구현 하기는 쉽지만 하나의 역할만 수행하는 모듈을 만들지는 못한다. 나중에 다시 그 역할을 맞추기 위해서 코드를 리팩토링하는 과정이 필요한 이유도 이것 때문이다. 역할이 많아지면 해야하는 일이 많아지며, 다양한 역할은 제대로 된 하나의 역할도 수행하지 못하도록 만든다. 모듈을 유지 보수하는 입장이라면, 하나의 변경이유에 대해서 한 모듈만 변경하는 것이 타당하다. 따라서, 여러가지 역할을 담당하는 모듈은 변경 이유도 여러가지가 되는 것이 당연하다.

모듈의 내부 자료구조가 외부로 드러나는 것은 모듈에 대한 통제 권한을 다른 모듈에 넘기는 것과 같은 일이다. 따라서, 모듈의 책임이 약화되는 결과를 낳게 되며, 내부의 변경에 대해서 외부의 코드 변경을 유발할 가능성이 높다. 따라서, 모듈은 전역적인 자료구조에 대한 사용을 가능한 줄여야 하며, 그러한 자료구조에 대한 접근 방법도 모듈에서 제공하는 인터페이스 만을 사용해야 한다. 모듈이 외부에 공개하는 인터페이스 들은 구조를 표현하는 것이 아니라, 요청을 받아들이는 창구 역할을 해야한다. 즉, 창구에서 받은 요청을 내부의 자료구조와 함수들을 이용해서 처리한 후, 결과를 알려주어야 한다. 만약, 구체적인 자료구조의 필드에 대한 접근을 허가한다면, 그것을 이용해서 처리해야 할 일이 해당 모듈 이외의 부분으로 전파될 가능성이 높다. 이러한 부분들은 모듈의 역할을 외부로 빼내는 결과가 될 것이다. 따라서, 모듈(Module)을 제대로 만들기만 해도, 대부분의 구현에서 구조적인 문제는 해결되었다고 볼 수 있다.

[캡슐화(Encapsulation)란?]

캡슐화가 좋다는 것은 이미 많이들 알고 있을 것이다. 한번 쯤은 다들 객체지향 코드를 짜기 위해서 C++이나 Java 등을 공부는 했을 것이고, 그렇기에 이런 개념들을 당연한 사실로 받아들인다. 하지만, 실제 자신이 구현한 코드에 대해서 객체지향에서 말하는 좋은 구조를 제대로 사용하고 있는가는 질문해 봐야 한다. 캡슐화는 내부 구현 정보를 외부로 최소한 만 보여주는 것을 의미한다. 실제로는 내부에서 정의하고 있는 각종 자료구조와 필드들에 대한 접근을 차단하는 것이 중요한 포인트다. 내부 정보를 접근하기 위해서는 제공되는 메쏘드(Method)만을 사용해야 하며, 제공되는 메쏘드 자체도 외부 정보에 대한 추측을 할 필요가 없어야 한다. 일반적으로 필드들에 대한 직접적인 값의 설정이나 얻어오는 메쏘드는 없어야 좋은 코드라고 할 수 있다. 이를 다른 언어에도 적용하면, 자료구조에 대한 숨김이 당연하게 받아들여져야 하지만, 생각보다 쉽지 않은 것이 사실이다.

사람은 위급한 상황에서는 가장 쉬운 해결책을 찾는 경향이 있다. 그것이 나중에 어떻게 될 것인가는 중요하지 않다. 현재의 위급한 상황을 넘기는 것이 먼저이기 때문이다. 따라서, 인터페이스를 위주로 하는 코딩이 아니라, 내부 구조라는 구체적인 것에 의존하는 코드를 만든다. 그것은 빠르게 일을 처리할 수 있다는 방어본능(급한 마음)이 발휘된 결과다. 특히, 시간을 다투는 상황이라면 이런 행동들은 지극히 당연하다는 공감까지 가세한다. 마치 "정보 은닉"이란 거추장스럽고 성능에도 도움이 되지 않는, 지킬 필요가 없는 코딩 습관으로 생각하게 된다. 이런 상황이라면 코드 리뷰나 단위 테스트와 같은 이야기는 너무 고상한 말로 들릴지도 모른다. 마치 교과서에는 있지만 실 생활에서는 전혀 도움이 안되는 먼 나라의 이야기처럼 생각될 것이다. 하지만, 여기서 "다름"이 생긴다는 것을 그때는 알지 못한다. 시간이 흘러 과제를 어렵게 마친 후 다시 원래의 코드로 돌아오면, 문제점들이 많다는 것을 발견하게 된다.



가장 큰 문제점은 코드가 변화를 수용하지도 못하며, 이해하기도 어렵게 되어있다는 것이다. 조금만 수정해도 버그가 발생하고, 그 버그를 수정하는 비용도 만만찮다는 것을 알게된다. 새로 팀에 들어온 인력들은 코드가 왜 그렇게 만들어 졌는지 이해가 부족한 상태에서 선불리 코드를 수정하지도 못하는 상태가 되고, 결국 기존 인력의 도움이 없이는 새로운 기능의 추가나 변경이 어렵다는 것만 깨닫는다. 물론, 기존의 인력들은 항상 높은 고과를 받고, 회사가 짜르기도 힘들지 모른다. 하지만, 스스로가 파놓은 함정에 더 깊숙이 발을 빼뜨리고 있다는 것은 깨닫지 못한다. 이미 좋은 기법이라고 말하는 것들을 도입할 시기가 늦었다는 것을 깨닫게 되고, 다른 제품을 개발하기 위해서 더 많은 지름길이라고 생각되는 방법들을

사용하게 된다. 사태는 해결보다는 누적되는 빚(Debt)으로 인해서 언제 파산할지 모르는 상황이 되어가고 있는 것이다. 해결은 단순하다. 스스로 자신의 코드가 잘못되었다는 것을 인정하는 것 밖에 없다.

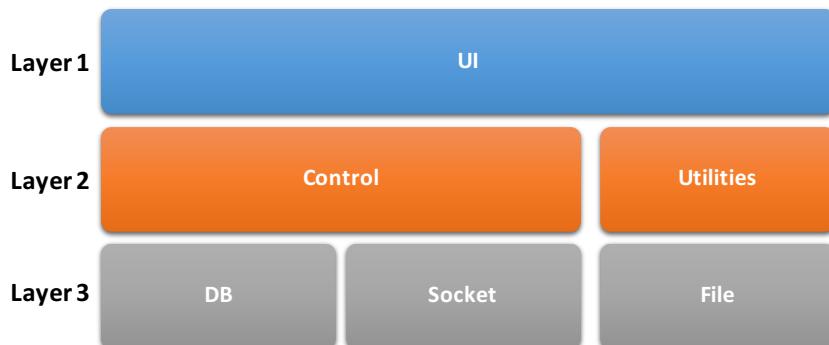
성능보다 항상 구조화가 먼저 되어야 한다. 구조화를 위해서는 구체적인 것에 의존해선 안되며, 내부의 정보를 가능한 적게 보여 주어야 한다. 구체적인 것에 의존하는 코드는 구체적인 것이 변경될 때 취약점을 보인다. 그리고, 내부 정보란 그런 구체적인 것들 중의 하나다. 따라서, 캡슐화는 의존성을 약화시키는 가장 효과적인 방벙이라고 볼 수 있다. 의존성이 약한 코드는 상대방의 변경에 대해서 영향을 적게 받는다. 자신이 변경되어야 할 이유도 단순하며, 변경의 영향을 특정 부분에 몰아넣게 된다. 캡슐화가 필요한 이유는 그런 변경이 상승효과를 발휘하지 못하도록 만드는 것이다.

자주 변경되는 코드가 있다면 그것은 잘못 작성 되었을 가능성이 높다. 수정하는 방법은 상호 의존성을 줄이는 것이며, 정보를 최소한만 공개하는 것이다. 성능을 위해서 직접적인 의존성을 늘리게 되면, 결국 유지보수 비용만 늘릴 뿐이다. 성능은 특정한 코드에 대해서 높일 수 있는 것이 아니며, 국지적(지역적) 이지도 않다. 구조적인 코드가 성능이 더 안정성을 보이는 이유도 이것 때문이다. 캡슐화가 되지않은 코드들은 서로 영향력을 주고 받기에 특정 부분만 고치면, 다른 부분에서 역효과도 같이 따라오게 된다. 사람과의 대화는 활성화를 해야하지만, 코드는 정보가 제한되어야 의존성을 줄일 수 있다.

[계층화(Layering)란?]

계층화란 모듈들이 계층(Layer)을 이루도록 역할과 책임을 세분화하는 것을 말한다. 즉, 시스템을 구성하는 모듈들의 상관 관계를 의존성을 이용해서 구분한 것이다. 계층적인 구조가 중요한 이유는, 계층을 통해서 역할을 각각의 모듈에 할당할 수 있다는 점이다. 상위 계층의 모듈은 하위 계층에서 제공하는 인터페이스를 이용해서 구현되어야 하며, 하위 모듈들은 그보다 더 하위의 모듈들이 제공하는 인터페이스에만 의존해야 한다. 이와 같은 의존성을 지키지 않는다는 것은, 상위 모듈과 하위 모듈의 역할에 구분이 없어진다는 말이며, 마치 하나의 모듈처럼 구현된다는 의미를 가진다. 따라서, 모듈의 역할에 중복이 발생하게 되며, 결국 변경에 대해서 강건한(Robust) 구조를 갖추지 못하게 된다.

계층화는 개념적으로 추상화를 통한 지식 쌓기의 일종이다. 즉, 구체적인 것에 대한 의존을 줄이며, 개념적으로 추상화를 이룬다는 말이다. 하위 계층에서 제공되는 인터페이스는 하위 모듈에 대한 추상화를 의미하며, 구체적인 것에 대한 의존을 줄이는 역할을 한다. 모듈 간에 계층화를 이루게 되면, 하위 모듈로 갈수록 구체적인 일을 하게되고, 상위 모듈로 갈수록 더 추상적인 역할을 맡게된다. 역할의 구분이 명확해지면, 각각의 계층에서 주로 담당해야 할 일이 정해지게 된다. 즉, 해당 계층에서 필요한 역할만 제공해야한다. 그렇지 않은 상황이라면, 계층 내부에 추상화 정도가 다른 모듈들이 섞이게 되며, 계층화를 통한 이점이 줄어들게 된다. 즉, 계층에서 처리해야 할 일의 수준이 달라져, 역할의 구분이 다시 모호해지게 되는 것이다.



나누어진 역할이라는 관점에서 추가적인 계층을 정의하는 것에 대해서 성능상의 이점이 줄어든다고 이야기 할지는 모르지만, 관리적인 측면에서는 변경의 유동성을 제어할 수 있게 된다. 계층화를 진행하면, 일종의 버퍼와 같은 것이 계층 사이에 존재하게 되며, 그 완충 역할이란 변화를 받아들이는 민감도와 연

결된다. 즉, 계층이 늘어날수록 구체적인 것이 추상적인 것에 영향을 덜주게 된다. 변경의 영향도가 감소됨에 따라, 코드는 가변적인 상황에 대해서 손쉽게 대응할 수 있다는 말이다. 줄어든 성능(함수의 호출 깊이)으로 변경에 대한 영향도를 감소시킬 수 있다는 것은 소프트웨어에 많은 이점을 준다. 즉, 같은 소프트웨어로 다른 제품에 다양하게 적용해야 하는 시점에는, 계층화를 통한 변경의 영향을 흡수하는 것이 필수적이기 때문이다.

변경은 소프트웨어가 가진 속성이다. 기능이 추가되거나, 버그를 고치는 것, 새로운 제품에 적용하는 것 등은 변경이 상시 발생할 수 있다는 것을 이야기 한다. 변경의 영향을 최소화하고 각각의 계층에서 제공하는 기능을 명확히 하는 것은, 역할과 책임을 구분할 뿐만 아니라 변화를 흡수하는 구조를 정의하는 것이다. 계층화를 위반하는 코드를 만드는 근본적인 이유는 이런 계층화에 대한 미흡한 약속을 과제 초기에 만들기 때문이며, 잘 만들어졌다고 하더라도 구현에서 그 원칙이 반영되지 않기 때문이다. 쉬운 해결책을 찾는 것은 짧은 시간의 만족을 주지만, 장기적인 만족을 위해서는 원칙이 우선시 되어야 한다. 그리고, 그 원칙이라는 것은 오랜 시간 증명된 것 이어야 한다. 계층화는 이런 원칙 중에서도 문제를 해결하기 위해서 오래동안 적용되어 왔던 것이라는 점은 누구도 부인하지 못한다. 하나의 계층이 맡은 역할과 책임을 다하는 것은 당연한 것이지 특별한 것이 아니기 때문이다.

[계층구조에 대한 단상]

계층적인 구조는 어떤 사물을 이해할 때 사용하는 "추상화(Abstraction)"가 기반이 된다. 즉, 상세한 구현을 모르더라도 그것을 사용할 수 있도록 만들어주는 방법이다. 계층화에서 어려운 부분은 사람의 실수가 항상 존재한다는 것이다. 즉, 상위의 계층은 바로 다음 하위 계층에서 제공하는 서비스만을 사용해야 하며, 하위 계층은 상위 계층으로 직접적인 접을 할 수 없다. 이를 위반하는 이유는 부주의한 코딩과 책임과 역할의 모호함에 기인한다.

한 가지 더 추가되는 것은 대부분의 모듈에서 공용으로 사용하는 부분을 어떻게 처리할 것인가를 정하는 것이다. 이때는 계층구조에서 특이한(예외적인) 부분이 있어야 한다는 것이다. 즉, 시스템에서 제공하는 인프라(Infra)와 유사한 형태의 "시스템 호출(System Call)"이나 기초 라이브러리(Library) 등을 위한 계층을 어떻게 정의할 것인지 도식화 하기 어렵다는 점이다.

사용할 수 있는 방법은 이러한 부분에 속한 모듈들을 모아서 하나의 계층으로 만들기 보다 옆으로 세운 형태로 정의하는 것이다. 공용으로 사용하는 계층도 규칙은 존재한다. 즉, 방향성을 주어야 한다는 것이다. 양방향으로 호출하는 구조는 바람직하지 않다. 단방향으로 어디서 어디로 호출이 발생하는지 정하는 것이 좋다. 즉, 구현하는 사람에게 어떻게 사용될지 알려주고, 구현된 것을 어떻게 사용할지도 파악하기 위해서는, 모듈과 모듈간의 호출 방향이 존재하는 것이 바람직하다.



구조적이지 못한 코드는 생각보다 비용이 많이 들며, 성능도 그다지 만족스럽지 못하다. 비용이란 결국 변화에 대응하는 것을 이야기 하는 것으로, 소프트웨어는 변경과의 전쟁이라고 해도 과언이 아니다. 소

프트웨어는 지속적으로 변경 되며, 변경되지 못하는 코드는 사라질 운명에 있는 것이 사실이다. 혹은 하나님의 제품에만 사용 되고 재사용 되지 못한다. 재사용 되지 않는 코드들은, 유사한 과제를 하더라도 이전 과제를 완료하는데 필요했던 시간이 걸릴 것이라는 말과 같다. 물론, 기존 과제에서 배운 것이 있을지도, 그것이 새로운 과제에 동일하게 적용될 것이라고는 장담하지 못한다.

계층이 많아지면 효율은 떨어지고, 대신에 구체적인 것과는 멀어진다. 따라서, 변경이 일어나는 부분은 낮은 계층에 집중하는 경향이 생긴다. 그리고, 분기되는 코드들이 놓이는 계층이 따로 생길 것이고, 그들을 위한 변경이나 통합 방안이 있어야 한다. 물론, 개별 과제들이 전부 다르다는 것은 인정하지만, 그렇다고 완전히 새로운 과제를 하는 경우는 드물다. 이 말은 기존 과제에서 어느 정도 유사성을 가진 과제를 할 가능성이 높다는 것이다. 예를 들어, 만약 특정 하드웨어가 있어야만 개발이 가능하다고 이야기 한다면, 이미 과제는 실패할 가능성이 높다. 소프트웨어 과제는 하드웨어를 가정해서 개발할 수도 있기 때문이다. HAL(Hardware Abstraction Layer)와 같은 것을 정의하는 것이 가능하기 때문이다.

계층화는 단순히 보기 좋게 만드는 것이 목표가 아니다. 보기 좋아서 이해하기 좋고, 수정하기 좋으며, 적은 비용으로 변경할 수 있도록 만드는 것이 목적이다. 계층화는 소프트웨어 발전을 보여주는 대표적인 예이며, 지식의 축적이 어떻게 이루어진다는 것을 보여 준다. 즉, 낮은 수준의 지식들이 모여서 새로운 토대를 이루고, 그 보다 높은 수준의 개념을 만들어나가는 것을 증명한다. 사람은 사물을 세세하게 파악하기보다는 뭉뚱그린 특징을 보기 좋다. 따라서, 계층화는 사람의 이해 능력에도 큰 영향을 준게 된다. 이는 인류가 지식이라는 것을 축적하는 과정에서 반드시 이뤄지는 과정이며, 소프트웨어라고 달라질 것은 없다. 오히려 역행하는 것이 이상한 것이다.

[데이터의 상태 변경에 대한 주의]

데이터는 읽는 것과 쓰는 것으로 구분되는 연산의 대상이다. 읽는 것은 일반적으로 변경을 발생시키지 않기 때문에 안정적이라고 생각되지만, 쓰는 것은 시스템의 상태를 변경하는 것 이기에 안정성에 영향을 주게 된다. 따라서, 시스템의 상태를 변경하는 것은 상당한 주의를 요구한다. 특히, 동기적으로 일어나야 하는 연산의 경우에는 어떤 순서로 상태 변경을 일으키느냐가 중요하다. 따라서, 이런 연산들은 적절한 곳에서 제어될 필요가 있다. 즉, 데이터를 읽는 곳은 여러곳에서 할 수 있지만, 데이터를 쓰는 것은 한 곳에 모아서 관리해야 한다.

개발자들은 전역 변수를 종종 사용한다. 물론, 전역 변수의 사용은 시스템 내부 모듈 들의 의존성을 높임과 동시에, 시스템의 변경 가능성과 유지 보수성에 악영향을 준다는 것은 다들 알고 있다. 하지만, 전역 변수가 주는 "사용하기 쉽다는 점"과 함수의 파라미터의 개수를 줄일 수 있다는 점은 언제나 매력적으로 다가오는 것이 사실이다. 그래서, 선택하는 이런 지름길이 결과적으로 코드를 더 복잡하게 만들어서, 기능 추가나 변경을 어렵게 만든다. 즉, 모듈 들이 데이터에 대해서 종속성을 가지게 되는 것이다. 쓰는 연산을 구현한 모듈 들은 읽는 연산을 수행하는 모듈 들에 대해서 영향을 주게되기 때문이다.

사실 전역 변수는 모듈들간의 결합성을 증가시키는 것 외에도 성능상의 문제점도 만든다. 즉, 전역 변수는 언제 값이 변경될지를 모르기에, 코드의 최적화에 악영향을 주며, 캐쉬를 이용하는 것보다 메모리를 직접 접근하게 만들어, CPU 사이클을 소모하게 만든다. 따라서, 전역 변수를 이용하는 것이 성능상에도 문제가 될 가능성이 있다. 개발자들이 생각하는 성능을 위해서 전역 변수를 사용한다는 것은 사실 별로 근거가 없다는 뜻이다. 따라서, 전역 변수를 가능한 최소한으로 줄이는 것이 성능에 더 유리하다고 생각할 수 있으며, 코드의 구조도 좀 더 깔끔하게 만들어 응집도가 높은 코드를 만들어 준다.

상태 변경과 상태 조회가 같이 일어나는 것은 좋지 않다. 그리고, 둘 이상의 상태를 한 번에 변경하는 것도 좋지 않다. 변경과 조회가 같이 있다는 말은 그것을 구현하는 코드가 둘 이상의 역할을 수행한다는 뜻이다. 즉, 변경하는 함수와 조회하는 함수는 나누어져야 하며, 그것을 호출하는 더 상위의 모듈에서 관리되어야 한다. 둘 이상의 상태가 변경되어야 한다면, 하나의 자료구조로 만들어 사용하는 것을 검토하는 것이 좋으며, 각각이 그런 식으로 묶여서 존재해야만 하는 이유가 타당해야 한다. 그렇지 않으면, 둘 이상의 상태가 각각 나누어서 변경될 가능성이 있으며, 이는 데이터의 일관성에 문제가 될 수 있다.

자료구조를 변경하는 것은 동기화(Synchronization)문제를 일으킬 가능성이 높다. 만약, 연결 리스트를 읽는 연산을 하나의 쓰레드(Thread)에서 수행하고, 연결 리스트를 조작하는 것을 다른 쓰레드에서 수행한다면, 둘 간에는 동기화 방법이 필요하다. 동기화되는 부분은 작을수록 좋으며, 가능한 가장 작은 범위에서 수행해야 한다. 동기화를 위한 순서도 중요하다. 연관된 동기화는 순서를 반드시 지켜야 할 필요가 있다. 데이터를 읽는 연산도 데이터를 변경하는 연산에 영향을 받는 상황에서는 동기화를 맞춰야 할 필요가 있으며, 이를 위해서는 가장 작은 단위의 동기화를 염두에 두어야 한다. 따라서, 데이터 자체에 대한 연산을 감싸는 새로운 함수를 정의할 필요가 있다.

데이터를 감싸는 연산을 정의해서 캡슐화(Encapsulation)하는 것은 데이터에 대한 접근에 일관성을 가지도록 만드는데 중요하다. 만약, 직접적으로 데이터에 대한 접근을 허용한다면, 누가 언제 데이터를 변경할지, 혹은 읽을지를 알 수 없게 된다. 물론, 단일 쓰레드에서도 이런 것은 문제가 될 수 있다. 즉, 인터럽트 서비스 루틴에서 부지불식 간에 사용하려고 하는 데이터를 변경할 가능성이 있기 때문이다. 공유되는 데이터에 대한 일관된 접근은 데이터를 일관되게 유지함과 더불어 자료구조에 의존적인 코드를 배제할 수 있기에, 모듈들 간의 독립성을 높여주는데도 도움이 된다.

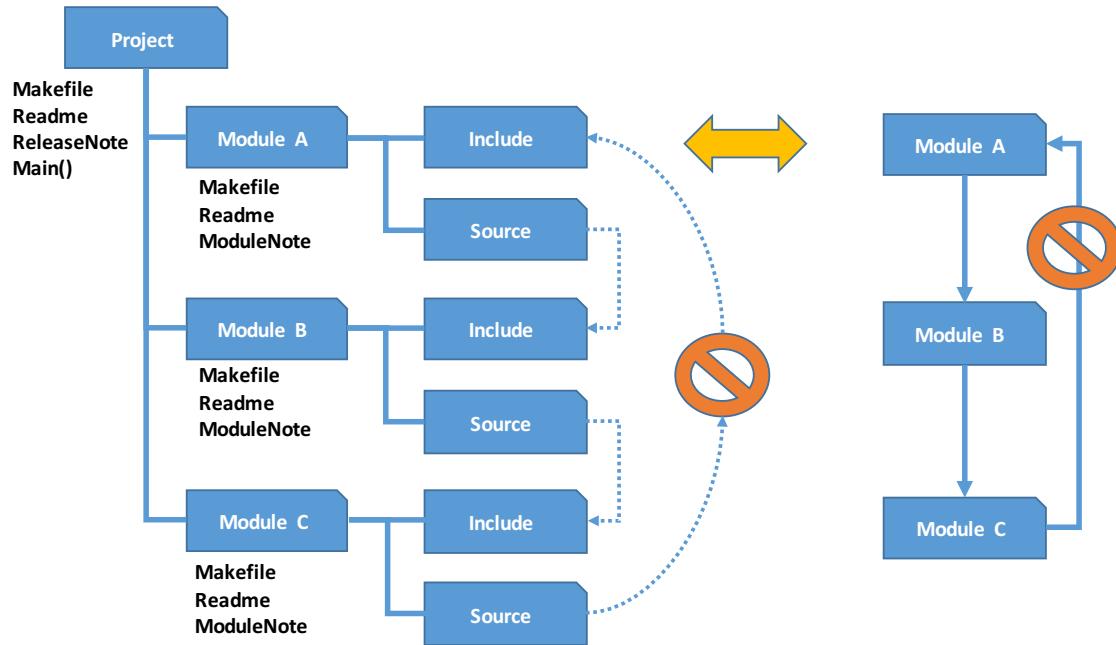
[디렉토리(Directory)]

프로그램을 작성하는 것은 창의력이 중요한 부분이 차지하지만, 프로그램을 유지하고 보수하는 일은 물리적인 제약 사항을 가지는 것이 유리한 경우가 많다. 또한, 훌륭한 구조를 만드는 일은 다양한 면에서 고려해야 할 것들이 있으며, 개발 초기부터 규칙을 정하지 않으면 개발 후반에 문제를 유발할 가능성이 높다. 따라서, 프로그램을 개발하기 위해서는 논리적인 구조의 설계나 상세 디자인만 중요시 할 것이 아니라, 물리적으로 폴더의 구조나 파일의 위치, 혹은 이름을 주는 방식 등을 고려하는 것이 유지보수에 많은 도움을 주게 된다.

예를 들어, 여러 개의 하부 모듈로 구성된 경우에 어떻게 디렉토리의 레이아웃(Layout)를 가지는 것이 좋을지 생각해 보도록 하자. 우리가 가진 모듈은 A라는 최 상위 모듈이 있고, 이 모듈 내부에는 하위 모듈로 a, b, c가 있다고 하자. 하위 모듈 각각은 다시 자신만의 헤더 파일을 정의하고 있으며, 공동으로 가져야 할 헤더 파일도 있다. 또한, 모듈 A를 위한 헤더 파일도 있다고 가정한다면, 전체 레이아웃은 아래와 같이 구성될 수 있을 것이다.

```
/Project/Module_A/source
/Project/Module_A/include
/Project/Module_A/Sub_Module_a/source
/Project/Module_A/Sub_Module_a/include
/Project/Module_A/Sub_Module_b/source
/Project/Module_A/Sub_Module_b/include
/Project/Module_A/Sub_Module_c/source
/Project/Module_A/Sub_Module_c/include
/Project/Module_A/Sub_Module/include
```

또한, 단위 테스트와 같은 것이 존재한다면, 단위 테스트를 위한 프레임워크(Framework)과 테스트 케이스를 가지는 디렉토리를 각각의 모듈 내부에 정의할 수도 있다. 따라서, 전체 파일들을 컴파일해서 하나의 실행 파일로 만들기 위해서는 각각의 모듈을 위한 빌드(Build) 스크립트(SCRIPT)를 따로 정의해야 하며, 이것들은 해당 모듈의 최상위에 위치하면 된다. 하위 모듈이 있다면 그것도 역시 포함해서 빌드시 포함되도록 만들어야 할 것이다.



상위 계층의 소스 코드는 하위 계층의 “Include” 디렉토리를 접근할 수 있지만, 하위 계층의 소스 코드는 상위 계층의 “include” 디렉토리에 대한 접근이 제한되어야 한다. 그렇지 않다면 계층간의 순환적인 의존 관계(Cyclic Dependency)가 생성되어, 한 계층의 코드 변경이 다른 계층의 코드에도 영향을 주게 된다. 이를 개선하기 위해서는 하위 계층에서 접근하는 상위 계층에 정의된 헤더 파일을 하위 계층으로 옮기거나, 헤더 파일의 일부를 새로운 헤더 파일로 정의해서 하위 계층의 “Include”로 옮겨야 할 것이다.

하나의 디렉토리 내부에 너무 많은 파일들이 있다면, 코드의 균형이 제대로 나누어진 것이 아니다. 물론, 그렇다고해서 버그가 있다는 것은 아니지만, 코드가 지나치게 치중된(비대한) 디렉토리가 존재하는 것은 역할과 책임의 명확한 구분이 되지 않았다는 의미가 될 수 있기에 주의해야 한다. 될 수 있으면 여러 디렉토리에 균형있게 코드가 분포하는 것이 더 좋은 선택이다. 파일의 내부도 마찬가지로 특정 파일에 지나치게 많은 코드가 들어가지 않도록 주의해야 한다. 즉, 하나의 모듈 내에 있는 파일들에 고르게 코드가 분포할 수 있도록 만들어 주는 것이 좋다.

[파일]

하나의 소스 코드를 담는 파일을 만드는 기준은 뭘까? 어떤 기준으로 만들어야 코드를 읽는 사람에게 조금이라도 더 많은 정보를 명확하게 제공할 수 있을까? 사실 코드를 작성하는 것보다 읽기 쉽게 만드는 것이 더 어려운 일이다. 누구나 코드를 작성할 수 있지만, “이해하기 쉬운 코드”는 항상 경험 많은 전문가만이 만들 수 있다.

먼저 소스 코드를 담는 파일을 만드는 자신만의 기준을 세우자. 그리고나서, 과제 전체에 대해서 그 기준을 꾸준히 준수하는 것이다. 방법보다 중요한 것이 일관성이다. 즉, 코드를 읽는 사람에게는 일관성이 더 많은 정보를 담을 수 있다. 예를 들어, 여기서 제시하는 방법은 외부 모듈에서 사용할 수 있도록 제공되는 함수에 대해서는 반드시 하나의 파일을 가진다는 원칙을 정했다.

따라서, 만약 “function_a()”있다면, 파일의 이름은 “function_a.c”가 될 것이다. 그외에 해당 함수가 사용하는 내부 함수들은 전부 동일한 파일에 저장하도록 한다. 하지만, 내부 함수라는 것을 알려주기 위해서는 반드시 “static”을 함수 정의와 함께 적어 주어야 할 것이다. 그리고, 함수의 이름도 마찬가지로 “_”와 같은 것을 붙여서 내부에서만 사용한다는 것을 나타내줄 수도 있다.

```
/* function_a.c file */
```

```

static void _internal_function( void ) {
    ...
    return;
}

void function_a( void ){
    ...
    _internal_function();
    ...
    return;
}

```

그럼 이렇게 했을 때의 좋은 점은 무엇일까 첫 번째는 파일의 이름으로 정의된 함수를 즉시 찾아낼 수 있다는 점이다. 그리고, 각각의 외부에 노출되는 함수들이 하나의 파일로 존재하기에, 뚜렷한 이유가 없이는 독립적으로 재사용 될 수도 있다. 내부 함수도 관련된 외부에 노출되는 함수와 함께 관리되기 때문에, 관련 있는 정보들을 모아서 관리하기 쉽다. 즉, 수정이 발생할 경우에 한 곳에서만 수정할 가능성이 높다.

헤더 파일은 여러 개의 외부에 노출되는 함수들의 선언을 가질 수 있으며, 혹은 헤더 파일들도 나누어서 만들 수 있다. 하지만, 지나치게 많은 헤더 파일이 남거나, 혹은 사용하는 자료구조가 공동으로 선언되어 있는 것이라면 하나의 헤더 파일을 이용해서 여러 개의 함수 선언들을 나열하면 될 것이다. 이때 조심해야 할 부분은 헤더 파일에는 자료구조의 내부를 정의하는 것은 가급적 하지 않아야 한다는 것과, 변수 선언과 같은 것은 제외하고 주로 타입에 대한 정의만을 담도록 해야 한다는 것이다.

헤더 파일내에서 다른 헤더 파일을 "#include"하는 것도 가급적 하지 않아야 한다. 이런 식으로 하면 헤더 파일들 간의 의존성이 커지는 결과가 생기게 되며, 컴파일 속도를 떨어뜨리는 원인이 될 수 있다. 따라서, 될 수 있으면 소스 코드를 가지는 파일에서 자신이 필요한 헤더 파일들을 순서에 맞게 "#include"하는 것이 좋다.

하나의 파일이 가지는 코드의 길이에도 제한을 설정하는 것이 좋다. 대략 1,000라인 정도면 충분할 것이다. 각각의 함수는 50(더 작아도 되지만)라인 정도 크기면 된다. 더 큰 파일이 필요하다면 파일을 나누는 것을 고려하거나 하위 모듈로 필요한 부분을 옮겨 놓을 수도 있을 것이다. 또한, 큰 함수가 필요하다면 작은 "inline" 함수 여러 개를 사용해서 큰 함수를 정의할 수 있다. 따라서, 이런 물리적인 제약도 지키면서 코드를 담는 파일을 생성한다면, 코드를 이해하는 측면에서 많은 도움을 얻을 수 있을 것이다.

[코드]

하나의 소스 파일은 어떤 구조를 가져야 할까? 가장 먼저 필요한 것은 소프트웨어의 라이센스에 대한 정보다. 즉, 지금부터 작성하는 소스코드의 소유가 어떻게 되는지 나타내는 헤딩(Heading)이 주석으로 들어갈 것이다.

```

/*
 * Copyright@2017, 2018 : This software has no license.
 * You can freely use this source code in whatever you want to do in your project.
 *
 * Author : suho.kwon@gmail.com
 * Description : xxxx
 */

```

다음으로 올 수 있는 것은 시스템에서 기본적으로 제공하는 헤더 파일들에 대한 "#include"문 들이다. 즉, 사용자가 정의한 헤더 파일보다 항상 먼저 선언해 주는 것이 좋다. 시스템의 헤더에 의존하고 있는

것들이 있다면 여기서 해결될 것이다.

```
#include <stdio.h>
#include <stdint.h>

#include "myheader.h"
...
```

헤더 파일들은 중복 포함이 되지 않도록 "#ifndef __XXX_H__" ~ "#endif"로 미리 처리되어 있어야 할 것이다. 또한, 헤더 파일이 다른 헤더 파일을 "#include"하지 않도록 주의한다. 즉, "#include"는 C 구현 파일 내에서 사용하는 것으로 한정하는 것이 좋을 것이다. 헤더 파일들간의 의존성을 가지는 것은 복잡한 관계를 만들어낼 가능성이 높기 때문이다.

헤더 파일의 포함이 끝났다면, 그 다음은 사용하는 자료구조의 형(Type)에 대한 정의를 가질 수 있다. 또한, 각종 상수 변수나 "enum"값들도 정의할 수 있을 것이다. 원칙은 사용되는 곳에 가장 가깝게 정의하는 것이지만, 전역적으로 사용되는 경우에는 이와 같이 상단에 위치하는 것이 좋다.

```
typedef struct my_struct {
    char *name;
    unsigned int age;
} MYSTRUCT;
```

```
typedef enum {
    JAN = 1,
    DEC = 12
} CALENDAR;
```

```
const unsigned int BIRTH_YEAR = 1971; /* rodata : readonly data */
```

대략적인 자료구조나 타입에 대한 정의가 끝났다면, 이제는 본격적으로 함수들을 나열할 차례다. 함수의 원형을 미리 선언해주어야 되지만, 그보다는 그냥 사용 되어야 할 함수를 먼저 정의하고, 그것을 사용하는 함수를 나중에 정의하는 방식으로 나열하면 된다. 따라서, 항상 함수는 선언되기 전에는 사용 되지 않을 것이다.

```
static unsigned int _get_age( MYSTRUCT *myInfo ) {
    return myInfo->age;
}
```

```
unsigned int get_person_age( MYSTRUCT *person ) {
    return _get_age( person );
}
```

내부에서만 사용되는 함수는 "static"으로 전부 처리해야 하며, 함수의 파라미터 중에서 변경되지 않는 부분에 대해서는 "const"로 선언해 주어야 할 것이다. 그리고, 당연히 외부에 제공되어야 할 함수는 자신에게 해당되는 헤더 파일에 따로 선언되어야 한다. 함수를 선언할 때는 단순히 타입만 명시하는 것이 아니라, 파라미터의 이름도 선언과 정의가 동일하도록 만들어주는 것이 실수를 유발하지 않도록 막아준다.

하나의 함수 크기는 50라인(혹은 30라인)을 넘겨선 안되기에, 그 이상의 라인을 가지는 함수를 발견하면, 내부 함수를 만들어서 쪼개주는 방법으로 정의하면 될 것이다. 이때의 내부 함수는 "inline"함수로 만

들기 위해서 "static inline"을 함께 이용해서 정의하도록 한다. 필요하다면 조건문이나 반복문 등을 "inline"함수로 만들어서 코드의 가독성(readability)를 높이는데 활용할 수 있다. 반복문 내부에서는 함수의 호출을 삼가해야 하며, 될 수 있으면 반복문을 포함해서 함께 내부 함수로 만들어주는 것이 성능에 유리하다.

[주석 사용법]

주석은 코드를 반복해서 설명하면 안된다. 즉, 코드를 통해서 알 수 있는 것을 설명할 필요가 없다. 코드와 주석이 동일한 내용을 설명한다면, 코드의 변경이 주석의 변경도 동반해야 하기 때문에 일종의 “복사된 것”이라고 볼 수 있다(“DRY:Do not Repeat Yourself”원칙). 주석으로 코드를 설명하려고 하지말고, 코드 자체가 주석이 필요하지 않도록 만드는 것이 좋다.

주석은 작성하기 쉽지만 관리하기는 어렵다. 주석이 많아지면 그것을 관리하기 위한 노력도 커진다. 주석으로 코드를 설명하게 되면, 코드와 주석의 내용이 일치하지 않는 현상이 자주 발생한다. 이때는 주석을 지우고, 코드를 더 쉽게 이해할 수 있도록 만들어야 할 것이다.

“관리되는 코드”에는 주석으로 처리된 코드가 없어야 한다. 코드가 주석으로 처리되어 있으면, 코드를 읽는 사람에게는 부담이 될 수밖에 없다. 남겨두면 지저분해 보이고, 지우려고 하면 다른 사람이 작성한 것을 임의로 변경하는 것 같기 때문이다. 이런 코드라면 그냥 지우는 것이 깨끗하다. 주석으로 남겨진 코드는 디버깅이나 과거 결정의 이유를 남기려는 목적이지만, 거의 다시 사용 되지 않는다. 관리되는 코드에는 주석으로 남겨진 코드가 없어야 한다.

주석에 누가 무엇을 했는지를 기록하지 말아야 한다. 그런 것들은 이미 버전 관리 시스템의 뒷이다. 시스템으로 충분한 일을 사람이 일일이 신경쓸 필요없다. 코드를 가능한 깨끗하고 읽기 쉽게 만드는 것은, 내용이 많아서가 아니라 알아야 할 내용이 적기 때문이다. 따라서, 반드시 필요한 주석이라고 생각되는 것 이외에는 다 지우도록 해야한다. 좋은 주석을 쓰는 방법은 아래와 같다.

01. 코드의 역할과 목적, 작성된 이유를 설명한다.
02. 일관된(정의한) 양식을 사용한다.
03. 필요한 내용만 짧고 간결하게 쓴다.
04. 영어로 쓴다. 한글로 쓰는 경우도 있지만, 해외 연구소와 협업을 위해서는 영어가 기본이다.
05. 복잡한 알고리즘과 같은 경우는 이해를 돋기 위해 원리를 자세히 적도록 한다.
06. 주석이 필요한 블록을 명확히 구분 되도록 만든다.
07. 주의 사항이나 문제점, 관련된 정보를 알려 준다.
08. 코드처럼 관리(리뷰)하도록 한다.
09. 코드 변경시 불일치가 있는지 “반드시” 확인 한다.
10. 불필요한 주석은 지운다.

주석은 코드의 일부라고 생각해야 한다. 주석도 코드와 같이 중복이 없어야 하며, 코드와 같이 관리되어야 한다. 잘못되거나 코드가 하는 일과 일치하지 않는 주석은 코드의 변경이 예상하지 못한 버그를 만들게 할 수 있다. 따라서, 주석을 많이 써서 코드를 더 관리하기 힘들게 만들기 보다 이해가 쉬운 코드를 짜고, 그래도 부족하다고 생각되면 이해를 돋기위해서 주석을 사용해야 한다.

[스타일]

코드를 잘 만드는 것과 그것을 이쁘게 만들어서 보여주는 것은 다르다고 생각할지도 모른다. 하지만, 코딩의 진정한 목적은 사용자의 요구를 만족시키는 것이 대부분이지만, 그 나머지는 코드를 읽는 사람들이 코드를 이해할 수 있도록 만드는 것도 포함할 수 있다. 코드를 읽는 목적은 코드를 이해하거나, 버그를 수정하는 것, 새로운 기능 추가와 같은 변경에 있다. 따라서, 조금이라도 읽는 사람들이 코드의 구조와 의미를 잘 파악할 수 있도록 만들어 준다면, 그 만큼 그 사람에게는 도움을 주는 일이 된다.

```

if( )
    /* K&R Style */
}

if( )
{
    /* BSD/Allman Style */

}

if( )
{
    /* Whitesmith Style */
}
...

```

코딩 스타일은 코드의 구조를 읽는 사람에게 알려줄 목적으로 사용한다. 개발자들은 자신만의 코딩 스타일을 사용하는 경우가 있지만, 일단 과제를 시작하면 과제에 관련된 모든 코드는 일관된 스타일을 사용해야 한다. 코딩 스타일에 대한 개인적인 선호가 존재하기에, 각자가 다른 코딩 스타일을 주장할 수 있다. 따라서, 중요한 부분만 정리가 된다면, 나머지는 그냥 툴에서 제공하는 방법을 그대로 사용하는 것도 한가지 방법이다. Eclipse같은 툴에서는 "K&R", "BSD/Allman", "GNU", "Whitesmith"등과 같은 코딩 스타일이 이미 정의되어 있으며, 단순히 파일 단위로 선택해서 적용해 주기만 하면 된다. 자동화된 적용은 사소한 버그를 잡아낼 수도 있기에, 적용한 후에는 자신의 코드에 로직상의 문제가 없는지 살펴봐야 할 것이다.

```

/* Make personal Information */
age = getAge();
weight = getWeight();

/* Update DB */
new_record = do_update_accout_information( name, age, weight );

/* Make a report for change */
do_change_report(new _record);

```

코드 내에 존재하는 공백은 마치 "동양화의 여백의 미"와 같은 의미를 지닌다고 볼 수 있다. 잠시 생각을 멈추고 새로운 시작을 알리거나 더 뚜렷하게 코드의 구조를 드러내 주기 때문이다. 적절한 공백의 사용은 코드를 읽는 사람에게 휴식을 주며, 시작과 끝이 어디인지를 나타내준다. 최소한 논리적으로 관련된 코드들은 공백을 사용해서 시작과 끝을 표현해주는 것이 읽는 사람에게는 잠시 생각의 쉼표를 찍을 수 있도록 만들어 줄 것이다. 이것도 마찬가지로 자동화된 툴을 이용하면 어느 정도 일관되게 보여줄 수는 있으나, 라인과 라인 사이의 논리적인 생각의 흐름은 개발자가 직접 처리해 주어야 한다.

```

unsigned int age;
unsigned int weight;
char *user_name;
float tax_rate;

```

스타일은 사람의 선호도를 반영하기에 정할 때 많은 논쟁이 있을 수 있다. 하지만, 그것으로 감정까지 상할 필요는 없다. 이미 표준으로 사용 되고 있는 스타일 들은 사람들간의 이러한 차이를 어느 정도 충분히 반영했다. 미세한 조정을 할 수 있다면 그것만 처리하고 일관되게 과제에 적용 하기만 하면 된다. 스타일

은 꾸미기며 원본의 의미는 바꾸지 못한다. 따라서, 잘 보이게 만들기 전에 코드 자체가 이해할 수 있도록 만들어 주어야 한다. 예를 들어, 변수 이름도 그냥 넘어갈 것이 아니라, 맑은 역할이 무엇인지 묘사적으로 설명할 수 있는 이름을 붙여주는 것이다.

[헤더 파일의 위치]

코딩을 하다보면 이미 선언된 헤더 파일이 필요한 경우가 생긴다. 문제는 이미 존재하는 헤더 파일의 위치가 코딩하고 있는 모듈의 위치와 다르며, 상위 계층의 모듈에서 정의하고 있는 경우다. 편의를 위해서는 그냥 상위 모듈의 헤더 파일을 컴파일러에서 불러올 수 있도록 만들고, 단순히 "#include"를 사용하면 될 것으로 보이지만, 이때 프로그램의 구조가 깨질 가능성이 있다.

먼저, 상위와 하위의 관계를 정의하도록 하자. 상위란 호출을 하는 측을 말한다. 당연히 하위란 호출에 대한 서비스를 제공하며, 호출 당하는 측을 말한다. 이때 구분이 힘든 경우가 있는데, 즉 서로 호출을 하고 호출을 당하는 관계가 있는 경우다. 이때, 더 많은 호출을 하고 있는 측을 상위에 있다고 정의하며, 더 적은 호출을 하고 있는 측이 하위에 있다고 본다.

모듈이란 함수들의 모임이며, 여러 개의 파일로 이루어진 경우가 일반적이다. 상위 모듈은 하위 모듈에 요청을 전달하기 위해서 하위 모듈에서 제공하는 함수나 인터페이스를 호출하게 되며, 하위 모듈은 그것을 만족시키기 위해서 호출을 처리하게 된다. 양방향의 호출 관계가 발생할 경우, 하위 모듈의 변경이 상위 모듈에 영향을 줄 가능성이 높아지게 되며, 이는 모듈간에 의존성을 높이고 코드의 변경 용이성(Modifiability)에 영향을 주게 된다. 따라서, 이런 상황을 제거하는 것이 모듈의 독립성을 높이는데 핵심이 된다.

Module A has Header File A

Module A calls Module B

Module B access Header File A

Module A <--> Module B

=====

Module B has Header File A

Module A access Header File A

Module A calls Module B

Module B access Header File A in Module B

모듈간의 양방향 의존성을 낮추기 위해서는 상위 모듈에서 하위 모듈로 가는 호출은 남겨두고, 하위 모듈에서 상위 모듈에 대한 참조(혹은, 호출)을 제거하는 것이다. 즉, 하위 모듈에서 필요한 자료구조나 함수는 하위 모듈에 위치시키는 방법이다. 상위 모듈에 정의된 자료구조라고 하더라도, 실제로 하위 모듈에서 사용된다면 상위 모듈에 존재해서는 안된다. 혹은, 두 개의 모듈을 하나로 합치거나, 상위 모듈에서 해당 자료구조를 사용하는 모든 함수를 하위 모듈로 옮겨야 한다.

함수의 위치나 자료구조의 선언 위치를 변경하는 것은 코드에 오류를 발생시킬 가능성이 낮다. 즉, 물리적인 이동이 있을 뿐 논리적인 변경은 없다. 하지만, 옮긴다는 행위에는 이런 부분만 들어가 있는 것은 아니다. 즉, 코드를 나누거나 파라미터의 추가 및 변경 등이 동반될 경우에는 오류가 발생할 수 있다. 따라서, 코드의 단순한 위치 변경이 아닌 경우에는 조금씩 변경하고 테스트하는 반복적인 과정이 필요하다. 이를 위해서 단위 테스트와 같은 것을 활용해 보는 것도 좋은 방법이다.

동작하고 테스트가 완료된 코드에 수정을 가하는 것은 일반적으로 위험하다고 생각한다. 물론, 맞는 말이다. 만약 해당 코드가 앞으로 전혀 변경될 가능성이 없고, 거의 완벽하게 테스트가 되었다고 보장된다면, 고치지 않는 것도 하나의 해답이다. 하지만, 만약 잊은 추가나 변경이 발생할 가능성이 있다면, 수정 요청시 구조적인 변경을 주는 것이 좋다. 이때도 한번에 많이 고치기보다는 조금씩 고치거나, 기존의 코

드를 남겨둔 상태에서 새로운 기능을 추가할 수 있도록 랩퍼(Wrapper) 계층을 추가해서 사용하는 것이 좋다.

대부분의 계층간 참조 위반은 헤더 파일의 위치에서 발생한다. 따라서, 헤더 파일을 같이 참조해야하는 하위 모듈로 이동시키는 것이 일반적인 계층화 방법이다. 더 좋은 것은 헤더 파일내에 정의된 정보를 상위 계층에서 알 필요가 없도록 인터페이스 역할을 하는 함수를 새로 정의하는 것이다. 즉, 하위 모듈에서 다른 내부 자료 구조를 상위 모듈에 노출 시키지 않는 것이다. 상위 모듈에서 필요한 것은 단순히 해당 자료구조가 무엇 인지만 인식하면 되며, 하위 모듈에서 실제로 내부 자료 구조에 대한 변경을 가하는 것이다.

[조건문 없애기]

조건문을 가능한 줄이는 것이 실행 속도를 높이고 코드의 구조를 개선하는데 효과적이다. 조건문은 복잡도(Complexity)와 관련되어 있으며, 코드의 이해를 방해하는 요인으로 작용한다. 따라서, 조건문이 많은 코드는 테스트하기도 힘들고 이해는 더욱 힘들다. 특히, 깊게 중첩된 조건문이나 여러개의 조건문의 조합은, 제대로 이해하기 위해서 생각보다 오랜 시간이 걸린다. 당연히 그렇게 만들어진 코드는 수정하기도 힘들다.

제품 모델의 분기를 위해서 조건문을 사용하는 것은 좋은 선택이 아니다. 이런 코드들의 특징은 지속적으로 커지는 경향(조건문이 추가되는)이 있다. 프로토콜(Protocol)을 처리하기 위해서 조건문을 사용하는 것도 좋지 않다. 그런 코드들은 대부분 아주 긴 라인으로 연결되어 있을 가능성이 높다. 따라서, 이런 부분들을 만나게 되면 다르게 처리할 수 있는 방법은 없는지 고민해 봐야 한다. 일반적으로 제품 모델의 분기에 대해서는 관련된 인터페이스 함수를 정의하고, 함수 포인터를 특정 제품을 위한 함수로 채우는 방법을 사용한다. 프로토콜을 처리하기 위해서는 프로토콜의 내부 정보를 이용해서 인덱싱(Indexing)할 수 있는 포인터 배열을 사용하는 것도 생각해 볼 수 있다.

```

typedef struct my_model {
    (void *function)( void );
    ...
} MY_MODEL;

enum MODEL {
    MODEL_A,
    MODEL_B,
    ...
};

MY_MODEL *model;

MY_MODEL *createModel( MODEL type ) {
    MY_MODEL *model;
    ...
    return model;
}

void call_model_specific_function( MYMODEL *model ) {
    ...
    /* Do something */
    ...
    model->fucntion(); /* Call the model specific function */
    ...
}

```

```

    return;
}

```

복잡한 조건문을 줄이기 위해서는 복잡한 조건이 나올 경우 "isXxxx()"-같은 짧은 "inline"함수를 사용할 필요도 있다. 혹은, 조건문 자체를 "드모르강의 법칙"과 같은 것을 이용해서 더 간단한 식으로 변경할 수도 있을 것이다. 이 때 미리 계산하기 이전의 원래 공식은 간단히 주석으로 남겨두는 것도 좋다. 그렇지 않다면 코드를 읽는 사람이 수정하기 쉽지 않은 조건문을 만들 가능성이 있기 때문이다.

```
if (!( x <= 100 )) ---> if ( x > 100 ) /* 물론, 이것은 간단한 예이다. */
```

조건문은 주로 참으로 표현되도록 만들어야 한다. 사람은 긍정적인 조건문을 더 쉽게 이해하기 때문이다. 조건문은 대부분의 조건을 만족시키는 경우 실행되어야 할 문장을 먼저 두는 것이 좋다. 드물게 실행되는 것을 나중으로 미뤄두는 것이 중첩된 조건문에 효과적이다. 예를 들어, "if(){} ~ else if (){} else {}"에서 첫 번째 "if()"를 더 자주 만족시킬 수 있도록 배치해 주어야 할 것이다. 두 번째 "if()"가 실행될 가능성을 최소화 하기 때문이다.

[자료구조 감추기]

프로그램에서는 자료구조를 숨기는 것이 의존성(Dependency)를 줄이는데 도움을 준다. 자료구조를 다루어야 하는 함수만 직접적으로 자료구조에 대해서 알 수 있도록 만들고, 그 함수를 호출하는 측에서는 자료구조를 몰라도 사용할 수 있도록 만들어 주는 것이다. 만약, 나중에 자료구조의 변경이 있을 경우, 해당 자료구조를 다루는 함수만 수정하면 되기 때문에, 사용하는 측에서 자신의 코드를 수정할 가능성은 줄어든다. 따라서, 동일한 인터페이스를 사용한다고 가정한다면, 독립적인 수정이 가능한 구조로 변경하는 것이다.

```

/* PrintInfo.h */
#ifndef PRINTINFO_H_
#define PRINTINFO_H_

typedef struct info INFO;

INFO *create_info(unsigned int age, unsigned int height);
void print_info(INFO *info);
void delete_info(INFO *info);

#endif /* PRINTINFO_H_ */

```

위의 코드는 자료구조를 다루는 함수와 관련된 타입에 대한 정보만 헤더파일에 정의했다. 즉, 구체적인 자료구조는 외부에 보여주지는 않고 있다. 따라서, 자료구조를 생성/조작/제거하기 위해서는 이곳에서 제공하는 함수만 사용할 수 있다. 함수를 사용하려면 이 파일을 "#include"해 주어야 한다.

```

/* PrintInfo.c */
#include <stdio.h>
#include "PrintInfo.h"

typedef struct info {
    unsigned int age;
    unsigned int height;
} INFO;

INFO *create_info(unsigned int age, unsigned int height) {

```

```

INFO *info;

if ((info = (INFO *) malloc(sizeof(INFO))) == NULL) {
    return NULL;
}
info->age = age;
info->height = height;
return info;
}

void print_info(INFO *info) {
    printf("Age : %d\n", info->age);
    printf("Height : %d\n", info->height);
    return;
}

void delete_info(INFO *info) {
    free(info);
    info = NULL;
    return;
}

```

자료구조의 정의는 구현파일에 넣어 두었다. 따라서, 외부에서는 직접적으로 자료구조의 각 필드를 조작할 수 없으며, 제공되는 함수를 사용하지 않고서는 아무 일도 할 수 없다. 일종의 외부 사용자와 계약을 맺은 것이라고 볼 수 있으며, 제공되는 함수 내에서 직접적으로 자료구조를 수정할 수 있도록 한정한 것이다.

```

/* ManageInfo.h */
#ifndef MANAGEINFO_H_
#define MANAGEINFO_H_

void do_manage_info(void);

#endif /* MANAGEINFO_H_ */

/* ManageInfo.c */
#include <stdio.h>

#include "PrintInfo.h"
#include "ManageInfo.h"

void do_manage_info(void) {
    INFO *info = NULL;
    info = create_info(47, 185);
    print_info(info);
    delete_info(info);
    return;
}

```

위의 코드에서 보듯이, 자료구조에 대한 직접적인 조작은 불가능하지만, 함수를 사용하기 위해서는 자료구조의 형태는 알아야 할 필요가 있다. 따라서, 자료구조의 형태와 그 자료구조를 다루기 위한 함수를 정

의하고 있는 "PrintInfo.h"를 "#include"해 주었다. 구체적인 자료구조의 필드를 알지 못해도 충분히 자료구조에 대한 조작을 주어진 인터페이스 함수를 통해서 할 수 있다는 것을 보여준다.

```
/* Main.c */
#include <stdio.h>
#include <stdlib.h>

#include "ManageInfo.h"

int main(void) {
    puts("Data Type Abstraction Example 01");
    do_manage_info();
    return EXIT_SUCCESS;
}

/* 실행 결과 */
Data Type Abstraction Example 01
Age : 47
Height : 185
```

코드의 실행 결과에서 프로그램은 의도한대로 동작한다는 것을 확인할 수 있을 것이다. 구체적인 자료구조를 직접 다루려고 하지말고, 그것을 다룰 수 있는 함수를 분리해서 만들어주는 것으로 변경에 대한 유연성을 확보한 것이다. 나중에 기능을 추가하거나 변경을 해야할 경우, 자료구조의 변경이 그것을 사용하는 측에는 아무런 영향을 주지 않고도 자유롭게 변경될 수 있다는 것을 알 수 있을 것이다.

물론, 성능 관점에서는 직접적으로 자료구조를 접근해서 수정하는 것이 가장 빠를 것이다. 하지만, 그런 코드는 변경에 취약한 것이 일반적으로 알려진 사실이다. 따라서, 일부 성능상의 오버헤드를 감수하더라도 더 좋은 구조를 만들기 위해서(변경 용이성, 유지 보수성)는 이런 식으로 의존성을 줄여주는 코딩을 하는 것이 구조적인 관점에서는 더 좋은 선택이 될 수 있다.

[전역 변수]

전역 변수(Global Variable)은 편리한 동시에 문제를 일으키는 원인이 될 수 있다. 따라서, 될 수 있으면 전역 변수를 사용하지 않는 것이 좋다. 편리하다는 것은 프로그램의 여러 부분에서 데이터를 직접적으로 가져다 사용할 수 있다는 것이다. 문제를 일으킴과 동시에 편리함을 제공해 주기에 경험이 미숙한 개발자들은 흔히 전역 변수를 많이 사용하게 된다. 하지만, 적절히 제한된 상황에서 전역 변수를 사용하는 것은 충분히 받아들일 수 있다.

예를 들어, 전역 변수를 사용하는 함수나 모듈을 한정하는 것이다. 즉, 관련된 몇 개의 함수만 변수를 공유하는 것이다. 변수의 공유가 늘어날수록 프로그램의 자유도는 떨어진다(하나의 블록 내에서 생각되어야 함, 서로 의존성이 높아짐). 공유되는 변수를 기준으로 함수와 모듈들이 의존성이 생긴다는 의미다. 의존성이 높은 프로그램은 변경이나 확장이 쉽지 않으며, 이는 결국 버그의 증가와 개발 기간의 연장을 가져오는 주요 원인으로 작용한다.

한정적으로 사용하는 것은 잘 나누어진 모듈이나 파일들로 구성된 하나의 디렉토리 수준으로 정할 수 있다. 이것이 미흡하다면 전역 변수에 접근하는 매크로(Macro)를 정의할 수도 있다. 이것도 좋지 않다면, 전역변수 수정을 전담하는 함수를 만들 수도 있다. 어쨌든 전역 변수의 사용을 최소화하고, 정해진 방법으로만 접근하는 것이 문제를 사전에 “지역화(Localize)”시킬 수 있는 중요한 방법임에는 틀림없다.

대량의 데이터를 서로 공유해야 하는 상황이라면 어떻게 해야할까? 당연히 포인터(Pointer)를 생각할 것이다. 하지만, 그것보다는 자료구조를 숨기는 것이 더 효과적일 수 있다. 즉, 자료구조와 그것을 다루

는 함수를 다른 모듈보다 낮은 계층으로 정의하고, 제공되는 API(Application Programming Interface)를 사용해서 자료구조를 접근하도록 만드는 것이다. 일종의 "칠판(Black Board)"와 같은 역할을 하는 것이 자료구조라고 보고, 다른 모듈들이 순서대로 칠판에 쓰거나 지우는 동작을 하는 것이다.

설정에 관련된 자료구조가 필요한 경우도 있다. 예를 들어, 프로그램 초기화 시에 읽어들인 자료를 나중에 다시 사용하는 경우를 보자. 만약, 함수의 파라미터(Parameter)로 전달한다면, 함수 호출의 경로가 길어질 경우 여러 번의 함수 파라미터 전달을 동반할 수 있다. 그렇다고 전역 변수를 이용해서 처리한다면, 경로 상의 모듈이나 함수들이 전부 의존성이 발생할 가능성도 있다. 이런 경우라면 위에서 제시한 방법으로 설정 데이터를 저장하는 자료구조를 정의하고, 다른 모듈 들보다 하위 계층에서 위치시키는 것이 좋을 것이다. 그리고, 초기화 시에 이용할 수 있는 방법을 제공하기 위해서 읽기와 쓰기에 해당하는 API들을 외부로 제공할 필요도 있다.

프로그램을 만드는 일에서 "지름길"은 비용(Cost)을 요구하는 경우가 많다. 소프트웨어 개발에서 비용은 "시간"이다. 즉, 뭔가를 하거나 하지 않아서 발생하는 비용은 그것을 없애기 위해서 시간을 필요로 한다. 그리고, 그 비용은 시간이 지나가면 복리이자로 불어나게 된다. 따라서, 문제가 발생하는 시점에서 고치는 것이 가장 비용이 싸다. 따라서, 전역변수의 사용 문제가 늙어져서 나오는 시점보다는 아예 전역 변수를 사용하지 않고 구현하는 것이 더 좋다.

만약, 성능상의 문제가 발생한다면 그 때 다시 전역변수를 사용해도 된다. 이미 다른 곳에서 그 변수에 접근하는 방법은 "제한"되어 있기에, 예외를 허락 하더라도 전역변수를 처음부터 사용한 것보다는 나을 것이 확실하다. 물론, 그렇다고 전역변수 사용을 과제 후반에 권장하는 것은 절대 아니다. 필요한 경우 사용을 검토 하라는 것이다.

[조건식(Conditional Expression)]

복잡한 조건식은 코드 이해를 방해한다. 조건식은 될 수 있으면 간단해야 하며, 몇 단계로 이어지는 조건식은 생각의 흐름을 끊어서, 버그를 유발할 가능성이 높다. 가장 좋은 코드는 조건식이 아예 없는 경우지만, 입력에 따라 실행을 변경해야 하는 경우는 어쩔 수 없이 발생한다. 따라서, 차선은 조건식을 간단히 만든데 집중하는 것이다.

```
if (( x > 0 ) && ( y <= -1 ) || ( z > 100 )) {
    ...
}
```

(주의!!! : 위의 코드는 컴파일러에 따라 경고 메시지를 출력할 수 있다. 컴파일러의 경고 메시지는 절대 무시해선 안된다. 사소한 실수가 심각한 오류로 이어질 수 있으며, 그런 버그들은 찾기도 힘들다. 기본적으로 모든 코드는 컴파일러 경고 수준을 가장 높여서 컴파일 되어야 하며, 모든 경고 메시지는 제거되어야 한다.)

위와 같은 조건식이 있다고 가정하자. 이것을 이해하기 위해서는 기본적으로 3가지 조건식이 의미하는 것이 무엇인지 알아야 한다.

```
if (( x > 0 ) && ( y <= 1 )) {
    ...
} else {
    if ( z > 100 ) {
        ...
    }
}
```

위의 코드는 어떤가? 위의 코드는 같은 일을 하지만, 앞의 코드보다 생각해야 할 코드의 조각이 나누어짐을 알 수 있다. 이처럼 코드를 작은 블록들로 나누어서 이해하면, 하나의 블록화된 코드 자체를 마치 추상적인 관점으로 묶어서 생각할 수 있게된다. 필요하다면 나중에 "if(z > 100){}"과 같은 부분을 다른 함수로 분리해 낼 수도 있을 것이다.

조건문은 간단하게 만드는 것은 코드의 이해를 돋는다고 했다. 대략적으로 2개 정도의 조건에 대해서는 이해의 어려움을 보이지 않는다. 하지만, 2개를 넘어서는 순간 대부분의 사람들은 앞의 조건을 기억해야 한다는 부담을 가진다. 따라서, 조건이 2개 이상을 넘어서지 않도록 해야할 것이다.

```
if (!( x > 0 )) {
    ...
}
```

```
if ( x <= 0 ) {
    ...
}
```

조건문이 부정을 나타내면 코드를 이해하는 것이 어렵다. 따라서, 부정적인 조건을 물어야 한다면, 긍정적인 조건으로 변경해서 사용하는 것이 좋다. 입력의 대부분이 처리될 수 있는 조건을 빨리 판정하는 것이 좋기에, 코드의 순서를 수정해도 상관없다.

```
static inline bool isNotAvailable() {
    ...
}

if ( isNotAvailable()){
    ...
}
```

```
static inline bool isAlreadyUsed() {
    ...
}

if ( isAlreadyUsed()) {
    ...
}
```

위의 코드는 부정적인 조건을 긍정적인 조건으로 바꾸어, 인라인(Inline) 함수화 시켜 사용하는 것을 보여준다. 보는 사람의 입장에서는 부정보다 긍정을 이해하는 것이 더 읽기 쉬운 코드를 만드는 비법이다. 따라서, 드모르강의 법칙과 같은 것을 이용해서 부정 조건문을 긍정 조건문으로 바꾸는 것도 의미있는 일이다. 물론, 세부적인 코드를 이해하는데 어려움이 생길 수 있다. 인라인 함수는 조건문에 이름을 붙여서, 상세한 구현을 감추면서 이해를 돋는 방법을 제공해 준다.

조건문이 복잡해지면 어려워지는 것은 테스트다. 즉, 조건에 따라 실행해야 할 경로(경우의 수)가 늘어나기에, 각각의 경우에 대해서 테스트 해야 할 케이스가 늘어난다. 경계값을 기준으로 했을 때, "x > 0"이라는 것을 테스트 하기 위해서는 "-1, 0, 1"이라는 적어도 3개의 값이 필요하다. 또한, 변수가 가질 수 있

는 최대/최소값도 케이스에 포함될 수 있다. 그것만 따져도 벌써 5개의 값을 넣어서 테스트 해야 한다는 말이 된다. 조건을 두 개 더 추가하면, 5의 3승 개수 만큼의 조건이 만들어질 수 있다. 물론, 그 중에서 상당 부분은 분석에 의해서 사라질 수 있지만, 어쨌든 늘어나는 것이 기하급수적으로 될 가능성은 충분히 있다.

```
#include <assert.h>

int add_two_value( int left, int right ) {
    return left + right;
}

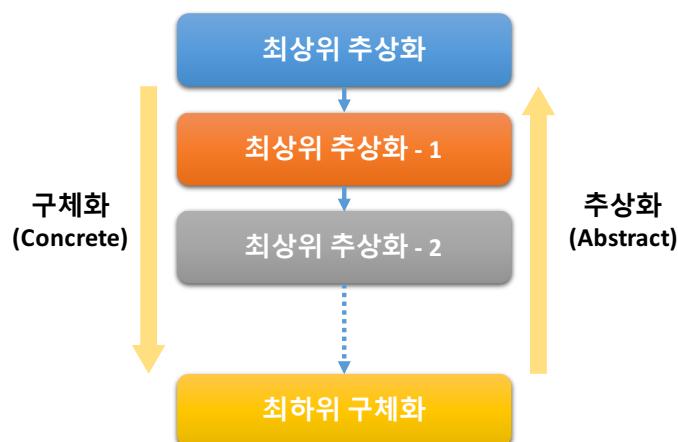
void main( void ){
    assert( 3 == add_two_value( 1, 2 ) );
    return;
}
```

따라서, 함수 단위의 테스트를 편하게 만들기 위해서도 조건은 될 수 있으면 간단한 것이 좋다. 복잡한 조건은 이해를 어렵게 만들 뿐 아니라, 테스트도 복잡하게 만들어 시간 소모도 늘어난다. 조금만 변경해도 그런 오버헤드를 없앨 수 있다면 충분한 개선 이유가 될 수 있다.

[추상화(Abstract)란?]

추상화란 구체적인 것을 감추고 보고 싶어하는 전체적인 특성(특징)을 드러내는 것이다. 소프트웨어 개발에서 추상화란 인터페이스에 의존하고 구체적인 구현에는 의존하지 않는 것을 말하며, 함수를 기본적인 추상화 방법으로 사용한다. 즉, 함수의 이름을 통해서 구체적으로 하는 일을 추상화 시켜서 나타낸다는 뜻이다. 예를 들어, "printf()"가 파일에 대한 출력을 담당한다는 의미로, 구체적인 출력에 관련된 내용을 추상적으로 대표해서 표현하는 것과 같다. 즉, 우리는 "printf()"가 실제로 출력에 대해서 어떤 일을 하거나 모르지만, 우리에게 무엇을 해줄지는 알고 사용한다. 그것으로도 구현하는데 충분한 정보를 주기 때문이다.

추상화는 생각의 단위를 묶어서 그룹으로 만들 수 있으며, 그것이 무엇을 의미하는지 알 수 있도록 해준다. 이것은 추상화를 지식을 표현하는 도구로 사용함과 동시에 이해를 돋는 방법으로도 사용하고 있다는 것을 보여 준다. 추상화가 잘 된 코드는 이름보다 한 단계 낮은 수준에서 내부의 구현을 구성하고 있다. 예를 들어, 함수는 함수가 가지는 이름보다 하나 더 낮은 추상화 수준에서 내부의 코드를 채워주어야 한다. 다시 내부의 코드로 더 깊이 들어가면, 그것 역시 다른 개념을 추상화시키고 있으며, 계속해서 가장 구체적인 내용이 나올 때까지 한 수준 씩 더 구체적으로 변화됨을 유추할 수 있을 것이다.



추상화가 제대로 되지 않은 코드는 추상화된 개념과 구체적인 개념들이 섞여서 같이 나오며, 이해하기 위해서 구체적인 것과 추상적인 것 사이를 오가게 만든다. 이런 코드를 본다면, 추상적인 부분은 그 수준이 맞는지를 검토해야 할 것이고, 구체적인 부분은 다시 다른 함수를 정의해서 추상화 시킬 수 있는 방법을 찾아야 할 것이다. 함수는 작게 만드는 것이 핵심이며 함수가 하는 일도 하나 이어야 한다. 그 하나의 역할이 함수의 이름으로 표현되며, 그 이름만 가지고도 무슨 역할을 하는지 명확하게 파악할 수 있어야 한다. 함수가 커지는 이유는 추상화를 게을리한 탓이며, 막연한 성능에 대한 기대 때문이다. 하지만, 함수가 커진다고 컴파일러가 최적화를 잘하는 것은 아니다.

상수에 대한 추상화는 이름을 붙이는 것이다. 상수를 그냥 사용한다면, 상수가 의미하는 것을 알 수 없게 된다. 이는 상수를 추상화 시켜서 지식으로 이해 되기를 바래는 행동이 아니다. 다양한 상수에 대한 정보들을 모아서 하나의 타입으로 선언해서 관리하는 것도 지식을 하나의 덩어리로 파악하기 위함이다. 어려운 조건문도 하나의 함수로 만드는 이유는, 조건문 하나 하나를 이해하는 노력보다 한번에 모아서 전체적으로 이해하기 위함이다. 함수 내의 블록화 된 코드들을 묶어서 새로운 함수로 만드는 이유도 블록에 대한 지식을 대표적으로 표현하기 위한 추상화다. 반복문 전체를 묶어서 함수로 만들어도 반복문이 무엇을 하는지 쉽게 이해하기 위해서다. 따라서, 추상화는 코드의 다양한 부분에서 적용해 볼 수 있는 기본적인 표현법이다.

파일의 이름이나 딕렉토리의 이름도 추상화의 일부이다. 즉, 파일의 내부를 뜯어보지 않더라도 그 파일의 역할이 무엇인지를 파악할 수 있으며, 딕렉토리의 내용물을 검토하지 않더라도 딕렉토리가 하는 일이 무엇인지를 알 수 있기 때문이다. 따라서, 좋은 이름을 만드는 것은 그 이름이 대표하는 추상화를 제대로 하는 것이다. 추상화에 실패한 과제는 제대로 정보를 묶어서 표현하지 못하는 경우이며, 그것으로 인해 구체적인 것에 대한 지나친 의존을 전체 시스템으로 퍼지도록 남겨둔 상태다. 추상화는 표현의 일부이며, 구체적인 구현은 표현과 분리되어 관리해야 한다. 함수의 이름과 파라미터의 변경없이도, 내부의 실제 구현은 변경될 수 있어야 한다. 따라서, 함수를 설계하는 것(이름과 파라미터 등을 정의하는 것)은 막 다른 골목에서 선택하는 어쩔 수 없는 행동이 아니라, 계획의 일부여야 한다. 함수는 가장 기본적인 추상화 도구이기 때문이다.

추상화는 프로그래밍의 중요한 도구이기에 그것을 제대로 사용할 수 있는 개발자가 되어야 한다. 결국 우리가 만드는 프로그램은 쌓아올려진 지식의 토대(추상화) 위에 숫가락을 하나 더 올리거나, 혹은 조금 더 추상화하는 것일 뿐이기 때문이다. 라이브러리를 쓴다거나, 특정 언어로 코딩하는 것 자체가 사물에 대한 형상을 추상적인 언어로 기술하는 행동이기 때문이다. 함수를 잘 만드는 것은 가장 기본적인 부품을 제대로 만드는 것이며, 추상화를 제대로 실천할 수 있는 대표적인 실천 방법이다. 함수가 길어지고, 이해되는 코드보다 이해되지 않는 코드가 늘어나는 것도, 추상화 실패에 원인이 있다. 따라서, 제대로 된 부품을 규격에 맞게 설계하고, 그것을 구현하는데 필요한 추상화 기술을 익히는 것이 그 해결책이다. 변경에 여유를 가지기 위해서는 구체적인 것에 의존해서는 안되며, 추상적인 인터페이스를 관리하는 것이 핵심이다.

[유지보수(Maintainability)를 쉽게 만들어주는 코딩]

어떤 언어를 사용하는지 상관없이 가장 중요한 코딩의 가르침은 "유지보수성(Maintainability)"이 높은 코드를 만들라는 것이다. 물론, 이렇게만 말하고 구체적인 방법을 이야기하지 않는 것은, 손짓으로 가야 할 방향은 알려주면서 어떻게 가야하는지에 대한 정보를 주지 않는 것과 마찬가지다. 지금부터는 그 방법을 이야기 하도록 하겠다.

1. 쉽게 읽히는 코드를 만들어야 한다.
2. 짧은 코드를 만들어야 한다.
3. 복제된 코드가 없어야 한다.
4. 하나의 기능만 충실히 구현해야 한다.
5. 테스트에 대한 자동화가 있어야 한다.

코드를 만들어 가는 원리는 "의존성을 줄이고(Decoupling)", "응집도(Cohesion)을 높이는 것"이다. 이 두 가지 원리는 각각 인터페이스와 모듈화로 나누어 볼 수 있으며, 이것들도 역시 유지보수를 쉽게 할 수 있는 코드에 맞춰져 있다. 마찬가지로 모듈 구현의 내부로 들어가면, 앞에서 본 것과 같이 유지보수성을 높이는 방법들로 코딩되어야 한다.

즉, 모듈을 개발의 최소 단위로(실행의 최소 단위와는 다름) 생각했을 때, 모듈이 외부와 연결되는 고리와 모듈의 내부를 어떻게 구성할 것인가에 따라, 코드가 쉽게 혹은 어렵게 유지보수될 수 있다는 것을 의미한다. 당연히 모든 개발자가 원하는 것은 쉽게 고치고, 기능을 추가하기도 쉬운 코드를 만나는 것이다. 자신의 코드는 이해하기 쉽다고 생각할지 모르지만, 남들에게도 그렇게 보여지는 것은 결코 아니다.

먼저, 유지보수성(Maintainability)에 대해서 간략히 알아 보도록 하자. 유지보수에는 수정과 기능 추가가 있다. 수정은 버그(Bug)나 구조적인 문제의 개선이 있으며, 기능 추가는 새로운 사용자의 요구사항을 기존의 코드에 넣는 것이다. 대부분의 코드는 사실상 개발 초기부터 유지보수를 하고 있다고 해도 과언이 아니다. 즉, 항상 새로운 기능을 조금씩 추가하고, 기존의 버그를 고치는 과정을 개발 완료까지 진행하기 때문이다.

이런 과정에서 생각한다면, 유지보수성을 고려한 코딩은 과제의 개발과정 전체와 개발 이후의 "유지보수"에 걸쳐서 일어나는 연속적인 활동으로 볼 수 있다. 물론, 코드의 성능도 중요하지만, 개발 과제의 성격상 가장 많은 비용이 들어가는 부분은 유지보수에 달려있다. 소프트웨어 개발 과제의 비용은 "시간"으로 환산이 가능하며, 비용이 많이 들어간다는 것은 시간이 추가적으로 필요하다는 말과 같다(과제 지연).

1. 쉽게 읽히는 코드

; 쉽게 읽히기 위해서는 코드가 서술적(Descriptive)으로 이야기를 해야 한다. 마치 소설 속의 모든 등장 인물들이 이름을 가지고 대략적인 성격을 파악할 수 있듯이, 좋은 이름을 가지는 것이 첫 번째 덕목이다. 변수, 함수, 모듈, 패키지, 디렉토리 등등 다양한 수준에서 좋은 이름은 읽기에 도움을 줄 수 있다.

2. 짧은 코드를 만들어야 한다.

; 코드는 짧을수록 이해하기 쉽다. 여기서 말하는 짧다는 기준은 코드의 라인(Line)수를 말하며, 라인수가 길어지면 하는 일도 많아지고 쉽게 복잡해 지기 마련이다. 전체적인 볼륨(Volume)으로서의 라인수를 말하는 것이 아니라, 프로그램을 구성하는 요소들의 라인수가 유지보수에서는 중요한 고려 사항이다. C언어의 경우에는 함수가 핵심이며, 대부분의 경우 30라인(최대 100라인 이하?)을 넘어서지 않아야 한다.

3. 복제된 코드가 없어야 한다.

; 복제된 코드는 모든 오류의 원천과도 같다. 코드 자체의 복제도 있지만, 논리(Program Logic)의 복제도 포함해야 한다. 복제를 없애는 것은 간단하지만, 의외로 잘 고치지 않는 것이 일반적이다. 예를 들어, 새로운 모델을 만들려고, 기존 모델의 코드와 비슷한 것을 복제해서 조금만 바꿔서 사용하는 것이 대표적이다. 이런 코드는 변경도 어렵고, 점점 더 버그를 유발하는 경향으로 발전할 가능성이 높다. "DRY(Do not Repeat Yourself)" 원칙을 꾸준히 유지해야 할 것이다.

복제된 코드를 검증하기 위해서 "CPD(Copy Paste Detector)"라는 툴을 사용할 수 있으며, 권장하는 기준은 0이다. 즉, 복제가 전혀 없도록 만들어야 한다. 물론, 이것이 힘든 경우도 있을 것이다. 예를 들어, ROM에 들어가는 코드는 NAND에 저장되어야 할 코드의 일부가 될 수 있고, 각각을 따로 코드 상에 유지하는 경우도 있다. 하지만, 잘 구성한다면 공통적인 부분과 달라져야 할 부분을 구분해서 나누는 것이 장기적으로는 더 좋은 유지보수성을 가지는 코드를 만들 것이다. 기준은 대략 6라인 이상이 동일한 코드가 있다면 복제되었다고 본다.

4. 하나의 기능만을 충실히 구현해야 한다.

; 기능을 많이 구현 하려는 욕심은 언제나 복잡한 코드로 이어진다. 다기능은 한 가지 기능에 문제가 생기면, 나머지 기능들도 제대로 동작하지 않을 가능성이 높으며, 기능간에 연결고리를 만들어 의존성을 높이게 된다. 차라리 인터페이스를 분리하고, 공용화 할 수 있는 부분을 뽑아내서 라이브러리 형태로 구축하는 것이 더 좋다.

간단한 구현 방법으로는 상태의 변경을 일으키는 것(Behavior)과, 상태를 묻는 것(Query)을 나누는 것 이 있을 수 있다. 예를 들어, 특정 상태를 얻어오는 것과 상태 변경을 발생하는 것을 각각 나누어서 함수로 구현하는 것이다. 상태 변경도 한 번에 한 가지만 하는 것이 좋다. 한 번에 한 가지 일만 잘 처리하는 함수를 만드는 것이 짧은 코드에도 도움이 되며, 파리미터의 개수나 테스트 케이스도 간단하게 만들어줄 것이다. 여러가지 기능을 한번에 묶어서 구현하지 말고, 기능별로 필요한 인터페이스를 제공하는 것이 좋다.

5. 테스트에 대한 자동화가 있어야 한다.

; 테스트는 자동화를 높여야 할 부분이다. 사람이 반복적으로 비슷한 일을 해야 할 경우라면, 자동화가 가장 빠른 해결 방법이기 때문이다. 물론, 모든 테스트를 자동화 시킬 수는 없다. 하지만, 단위 테스트나 통합 테스트, 시스템 테스트의 일부는 자동화가 가능하다. 자동화의 이점은 과제의 후반으로 갈수록 더 높아지만, 문제는 테스트 케이스 작성을 쉽게 하기 위해서는 앞에서 이야기한 네 가지를 잘 지켜져야 한다는 것이다.

잘 이해도 되지 않고, 라인 수도 많으며, 하는 일도 많은 함수는 테스트 하기도 힘들다. 즉, 테스트 케이스를 더 많이 만들어야 하며, 테스트 되지 않는 코드가 많아질 가능성이 높다. 코드 간의 의존성이 높아서, 테스트 자체보다는 부가적으로 구축해야 할 것들도 늘어나게 된다. 따라서, 생산성도 낮아지게 되며, 테스트 신뢰도도 같이 낮아진다. 앞에서 열거한 것들을 충실히 지키기 위해서 “테스트 커버리지(Test Coverage)”와 같은 것을 점검할 수 있다.

개인적인 생각으로는 대략 90%이상의 코드가 테스트 커버리지에 속한다면, 테스트 자동화 수준이 괜찮다고 볼 수 있다. 대부분의 과제에서 자동화 테스트는 대략 50%~60%정도의 수준에 머물기에, 더 많은 버그가 수동 테스트를 거쳐야 발견될 것이다. 당연히 비용은 상승하고, 피드백은 느려질 것이다. 과제 기간에 대한 예측도 불확실성이 높아지게 된다.

위에서 나열한 방법들에 대한 반론도 있다. 예를 들어, 함수의 라인수를 줄이면 함수 호출 횟수가 늘어나서 성능에 악영향을 줄 수 있다는 것이 대표적이다. 하지만, 함수의 라인수가 늘어난다는 것은 결국 많은 일을 하고 있다는 뜻이되며, 함수가 넘겨받는 인자나 내부의 지역변수의 수가 늘어난다는 뜻이다. 즉, 변수들이 레지스터를 이용하는 것보다 스택에 할당될 확률이 높아서 메모리 접근 시간이 추가적으로 더 필요하며, 전역 변수를 쓴다면 함수의 최적화(당연히 메모리 접근도 늘어남)도 문제가 될 수 있다. 또한, 함수 자체가 길이가 길기 때문에, 코드의 중복이 발생할 가능성이 높고, 재활용이 안되기에 컴파일된 실행 파일의 크기를 증가시킬 가능성도 있다.

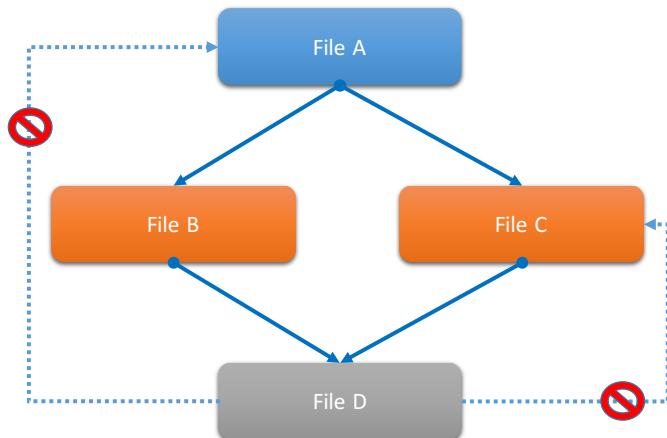
따라서, 함수의 크기를 작게 만드는 것이 크게 만드는 것보다 유리한 점이 더 많다. 컴파일러도 결국 사람이 만들기에 복잡한 함수를 최적화 하는데 한계를 가지며, 함수의 호출 오버헤드도 최적화의 중요한 고려 요소이기에 많은 신경을 써서 만들어 졌다. 함수를 크게 만들어서 최적화를 한다고 이야기하는 것은 올바르지 않다. 물론, 루프와 같은 반복문에서 잦은 함수 호출은 성능 문제를 유발 할 수 있다. 그런 경우에는 호출되는 함수 내부로 루프를 이동시킬 수도 있다.

[파일의 구성]

C언어는 크게 두 가지 파일을 사용한다. 하나는 각종 데이터의 형(Type)을 지정하는 헤더(Header: ".h")파일이며, 실행할 코드를 자기는 ".c"로 를 확장자(Extension)를 가지는 파일이다. 각각의 ".c"파일은 자신이 사용하는 데이터의 형에 대한 것을 정의하기 위해서 ".h"파일을 하나 이상 가질 수 있으며, 각각의 ".h"파일도 마찬가지로 ".h"를 하나 이상 가질 수 있다. 대부분의 경우, 하나의 ".c"파일은 하나의

".h"파일을 자신만을 위해서 사용하며, 만약 필요한 다른 헤더파일이 있다면, 그것을 가져올 수 있다. 즉, 고유하게 ".c"파일의 데이터 타입은 하나의 헤더파일을 가지는 것을 주로 많이 사용한다. 하나의 헤더파일이 여러 개의 다른 헤더 파일을 불러와서 사용할 수도 있지만, 이럴 경우 헤더 파일들 간의 의존적인 관계를 만들어서, 프로그램 수정에 악 영향을 미칠 수 있으며 컴파일 시에 시간이 더 많이 걸리게 만들 수 있다. 따라서, 필요한 헤더 파일은 주로 ".c"에서 직접 다른 헤더파일들을 가져오는 것이 좋다.

파일들간에 서로 참조하는 경우가 많아질수록 프로그램은 이해하기는 어려워지는 경향이 있다. 따라서, 이런 것들을 방지하기 위해서는 파일의 이름을 결정할 때, 역할을 알려줄 수 있는 이름을 주고 그 역할에 해당하지 않는 코드를 담아두지 않는 방법을 사용한다. 하지만, 코딩을 진행해나가면서 이런 규칙들이 조금씩 무너지게 되는데, 이때는 역할에 따른 분류 및 관계를 따져서, 새로운 파일로 만들어 주거나, 기존의 다른 파일로 함수나 자료구조, 변수등을 이동시켜 주도록 한다. 원칙은 "사용하는 곳에 정의한다"를 따르면 될 것이다. 예를 들어, "A.c"에 정의된 함수가 "B.h"에 정의된 자료구조를 접근한다면, 그 함수는 "B.c"로 이동하는 것이 맞을 것이다.



파일들 간의 상호 작용은 적게 만들어야 한다. 즉, 다른 파일에 선언된 함수에 많이 의존할 수록 코드는 의존성이 강해진다. 의존성이 강한 코드는 변경에 약하며, 새로운 기능을 추가하거나 버그를 수정 하더라도 또 다른 문제를 일으키게 된다. 따라서, 가능하면 적은 수의 함수들을 외부에서 볼 수 있도록 선언해야 한다. 또한, 데이터가 있는 곳에 함수를 정의하는 것이 좋다. 즉, 데이터를 다루는 함수는 같은 모듈에 두는게 이해도 쉽고 명확한 역할 구분을 돋는다. 예를 들어, 사람에 대한 자료구조를 정의한 곳에서만 정보를 설정하거나 다른 모듈에 있는 함수를 가질 수 있도록 만드는 것이다. 만약 다른 쪽에서 사람에 대한 정보가 필요한 경우에는 반드시 외부에서 보이는 함수들을 통해서 그 정보에 접근하도록 만드는 것이다.

프로그램의 구조적인 문제는 대부분 특정 모듈과 외부의 연결에서 발생한다. 따라서, 이를 해결하기 위해서 코딩 시에 원칙을 가지고 접근하는 것이 좋다. 가장 쉬운 원칙은 자신이 사용하는 것과 자신을 사용하는 코드를 명확히 구분하는 것이다. 즉, 두 가지를 각각 하나의 집합으로 생각할 때 교집합이 없도록 만드는 것이다. 이렇게 하게되면 "A->B-C->A"와 같은 호출의 사이클(cycle)이 만들어지는 것을 방지해 줄 수 있다. 상호 의존성을 가지는 코드는 하나의 변경이 다른 하나의 변경으로 이어질 수 있기에, 코드를 유지보수하기 어렵게 만든다. 따라서, 단방향 의존성을 가지는 코드를 생성하는 것이 구조를 단순화시키는 방법이다. 만약, 상호 의존성이 생기는 모듈이 있다면, 각각의 모듈이 맡은 역할에 혼돈이 있다는 뜻이 되며, 모듈의 역할을 명확히하고 역할에 부합하는 함수들만 남길 수 있도록 해야한다.

마지막으로, 남아있을 수 있는 문제는 헤더 파일들간의 상호 참조다. 이것에 대한 해결책은 될 수 있으면 헤더 파일에서 자신을 필요로 하는 ".c"파일에서 사용되는 자료구조의 타입만 정의하는 것이다. 만약, 다른 헤더 파일에서 정의된 타입이 필요하다면, 그것을 "extern"과 같은 것을 사용해서 링크 시(Link-Time) 해결할 수 있도록 만든다. 헤더 파일의 의존성도 다른 파일들과 마찬가지로 중요하게 관리해주어야 한다.

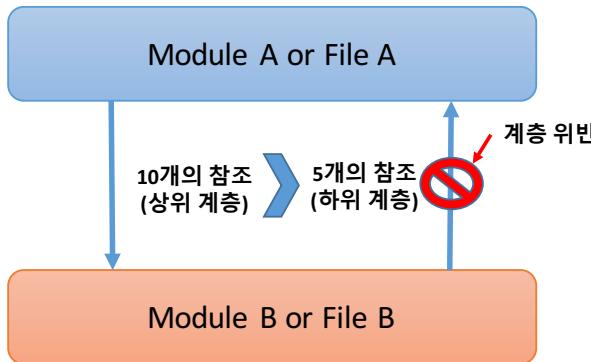
야 할 것이다. 헤더 파일의 자료구조도 분리할 필요가 있다면, 각각 다른 헤더 파일로 만들어 해당하는 모듈로 옮겨주어야 한다.

[파일간의 의존성 낮추기]

파일들간의 의존성을 제거하는 것은 중요하다. 즉, 하나의 파일이 의존하고 있는 파일은 어떤 것들이 있는지 파악할 수 있어야 한다. 보통 코드가 꼬이는(스파게티 코드가 되는) 경우는 하나의 파일이 여러개의 파일에 의존하며, 또한, 그 중 몇몇 파일들이 순환을 이루어 원래의 파일에 의존하는 경우다. 이때 순환을 이루는 연결에 참여한 특정 파일을 수정하는 것이, 다른 부차적인 효과(Side-Effect)를 가져오게 만들 수 있다.

될 수 있으면 의존성을 줄이는 것이 중요하다. 즉, 하나의 파일이 다른 파일에 접근하는 것을 최소화 하는 것을 말한다. 이를 "Fan-Out"이라고 이야기하며, 하드웨어에서 특정 IP의 출력이 다른 IP들의 입력에 몇 개의 연결을 가지고 있는지를 차용한 것이다. 반대로 자신이 얼마나 다른 모듈에서 사용되고 있는지를 알려주는 "Fan-In"도 있다. 의존성을 줄이면서 효과적으로 파일들을 구조화 시키기 위해서는, Fan-Out은 줄이고 Fan-In은 늘리는 방법을 사용하게 된다.

"Include"를 통해서 참고하는 헤더 파일들도 많으면 많을수록 문제가 될 수 있다. 더군다나, 헤더 파일들 간에도 "Include"를 사용해서 참조하는 경우도 문제가 될 수 있다. 즉, 하나의 헤더 파일을 수정하면, 그것을 참조하는 다른 헤더 파일들에 영향을 주게된다. 따라서, 결과적으로 구현 파일인 C파일들도 같이 영향을 받는 것이다. 따라서, 이런 상황에서 참조가 필요한 헤더 파일들은 가능한 C구현 파일에서 "include"하는 것이 바람직하며, Header 파일 내에는 가능한 적은 정보를 가지도록 만들어야 한다. 구현 파일에서 참고해야 할 데이터를 줄이면 줄일수록 의존성은 약해진다.



파일들 간에는 반드시 단방향(Uni-Directional)의 의존성만 존재해야 한다. 상호 의존적인 파일들은 문제를 일으킬 가능성이 높다. 지금은 문제가 안되더라도 나중에 코드의 수정이 발생할 때 문제가 발생할 수 있다. 추가적으로, 간접적인 상호 의존성을 만드는 경우도 제거되어야 한다. 즉, 여러개의 파일이 순환적인(Cycle) 참조 의존성을 만들어내는 경우다. 사이클이 생성되면 그 사이클에 참여하는 모든 파일들은 서로 영향을 줄 가능성이 높으며, 계층구조나 코드의 재사용성에 영향을 미치게 된다. 따라서, 그런 것들을 숙아내서 역할과 책임에 맞게 코드를 재배치하는 것이 해결 방법이다.

C구현 파일 하나에 하나의 헤더 파일을 두는 것이 좋다. 헤더 파일에는 다른 헤더 파일을 포함시키지 않지만, 예외적으로 포함할 경우도 있다. 하지만, 포함되는 헤더 파일들의 연속된 고리가 길어지지 않도록 유지해야 한다. 필요한 헤더 파일들은 C구현 파일에서 직접 "include"하는 것이 좋다. 만약, 특정 자료형(Data Type)이 필요한 경우에는 자료구조의 내부까지는 포함하지 않도록 해야 한다. 직접적으로 해당 자료구조의 내부를 접근하는 것은 막아야 하며, 가능한 외부로 공개되는 정보를 줄이는 것이 프로그램의 유지보수나 확장에 유리하다. C구현 파일에서도 외부에서 접근할 필요가 없는 함수들은 특별히 관리하는 것이 좋다. "static"을 사용해서 파일 범위에서만 유효하도록 만들어 주어야 한다.

만약, 다른 파일의 함수를 접근할 필요가 있다면, 상대방 파일에서 역으로 접근하지 않도록 만들어야 한다. 이런 구조가 흔히 발생하는 이유는 제대로 파일들의 역할을 구분하지 않았기 때문이다. 그런 경우가 생기면, 함수 자체를 이동시키거나 새로운 함수를 만들어 단방향 접근으로 한정 하도록 한다. 필요한 자료구조가 다른 파일(모듈)에서 정의하고 있다면, 해당 헤더 파일을 C구현 파일에서 "include"해서 사용하도록 만든다. 이때도 물론 자료구조에 대한 직접적인 접근보다는 "Access"함수와 같은 것을 정의해서 사용하는 것이 변화에 둔감한 코드를 만드는 방법이다. 자료구조의 타입이 필요한 경우는 자료구조의 이름만 사용하도록 한정 한다.

의존성이 발생하는 것은 어쩔 수 없지만, 의존성에 방향성이 없는 상태라면 문제가 있다고 생각해야 할 것이다. 그리고, 그 의존성이 순환 구조를 만들면 수정할 필요가 있다. 이때 한가지 예외적인 상황이 있는데, “콜백(Callback)”을 사용해 특정 이벤트(Event)가 발생했을 때 호출하는 함수를 미리 설정해 준 경우다. 하지만, 이것은 순환적인 참조를 발생시키는 것이 아니며, 외부에 의존하는 코드도 아니다. 즉, 콜백을 구현하기 위해서 미리 정의된 인터페이스를 가정 하기에, 그것에 맞도록 호출만 하는 것이다. 따라서, 호출되는 코드와 의존성은 인터페이스로 한정된다. 따라서, 순환구조로 생각하지 않아도 된다.

파일 간의 상호 관계는 설계에서 보여주는 의존성을 반영해야 한다. 즉, 설계에서 만든 그대로 의존 관계가 성립해야 한다. 만약, 그렇지 않다면, 설계를 수정하거나 코드를 다시 작성해야 할 것이다. 설계와 코드간의 괴리가 발생하면, 나중에는 둘 중 하나를 무시하게 되며, 당연히 문서를 믿지 못하게 될 것이 분명하다. 따라서, 지속적으로 사용 되는 문서를 만들고자 한다면, 문서도 항상 유지 보수가 되는 형태로 관리해야 할 것이다. 하나의 모듈은 하나의 디렉토리를 차지하는 것이 좋으며, 모듈 자체로 컴파일과 설치를 할 수 있으면 좋을 것이다. 물론, 다른 모듈을 의존하는 관계를 내부에 가지고 있다면, 의존하고 있는 모듈도 실행하는데 필요하다. 모듈과 모듈간의 연결관계는 오류가 발생하는 근거가 될 수 있으므로, 그 관리를 철저히하는 것이 좋다. 즉, 너무 많은 외부 인터페이스를 제공하는 것은 오류를 많이 발생시킬 가능성이 높다. 그렇다고, 모든 파일을 하나의 디렉토리에 두는 것도 관리적인 차원에서 좋은 방법은 아니며, 디렉토리 내부의 파일들이 하나의 거대한 코드 덩어리가 되는 경향이 생긴다.

[한 줄에 하나의 명령어만 사용]

될 수 있으면 한 라인에는 반드시 하나의 명령이 있는 것이 좋다. 흔히, 조금 개발에 익숙한 개발자들은 짧은 문장을 구성하려고, 언어의 문법에서 제공되는 기능들을 "고차원(?)”적으로 사용하지만, 그렇게 한다고 좋은 코드가 만들어지는 것은 아니다. 즉, 한번에 처리할 수 있는 명령어를 하나씩 두는 것이 좋다. 변수의 선언이나 할당문과 같은 곳에서 여러 문장을 사용하는 것은 자칫 오류를 만들어 낼 가능성이 있기 때문이다.

```
int a, b; --> int a;
           int b;

a = (a++) + b; --> a = a + b;
           a++;
```

이런 것들은 어려운 것은 아니지만, 가끔 오류가 발생하는 이유는 코드를 수정하는 사람이 부주의해서 실수를 할 수 있기 때문이다. 따라서, 간단한 문장을 만들어 의도를 명확히 표현해 주는 것이 좋다. 사실 코드에서 발견되는 많은 찾기 어려운 버그들은 이처럼 간단한 실수에서 종종 발생한다.

```
struct myStruct * a;
int b;

(*(++a)).next = b; --> ++a;
a->next = b;
```

포인터와 같은 경우에도 짧게 만들려고 "+"나 "-"를 사용할 수 있는데, 이때도 마찬가지로 코드의 의도를 명확히 한 줄씩 나누어서 적어주는 것이 좋다. 실제로 이렇게 만들어도 어셈블리(Assembly)어는 동일하게 나올 것이다. 한 줄로 만든다고 원자적인 연산(Atomic Operation)이 가능한 것도 아니다.

```
int a;
int b;
int c;

a = a + b / a - c; --> a = ( a + b ) / ( a - c );
```

코드가 간결해야 한다는 것은 짧게 코드를 만들라는 의미보다, 코드가 프로그래머의 의도를 코드를 읽는 사람에게 명확히 전달할 수 있어야 한다는 의미다. 코드를 읽는 컴파일러를 위한 것이 아니라, "사람"을 위한 것이다. 사람이 이해하기 쉬운 코드를 만드는 것이 좋은 코드를 만드는 지름길이다.

[조건문에 할당문이 들어가지 않도록 주의]

조건문을 판단할 때는 사소한 에러가 발생할 수 있다. C언어는 C#과는 달리 "if()"와 같은 조건문의 괄호에 수식을 가질 수 있으며, 그 값이 0인 경우를 제외하면 모두 참으로 생각한다. 따라서, 아래와 같은 문장도 가능하다.

```
...
if( x = 10 ) {
    ...
}
```

물론, 이렇게 해야하는 경우도 있다. 하지만, 이것이 정말 프로그래머가 의도한 것인지 다시 한번 생각해 봐야할 것이다. 따라서, 위의 예에서 프로그래머의 의도는 아래와 같을 것으로 생각한다.

```
...
if( x == 10 ) {
    ...
}
```

즉, "x"라는 변수의 값을 판단하기 위해서 사용한 것이 "if()"문이지, 변수 자체의 값을 이용하기 위한 것 이 아니다. 사실, 이런 오류는 흔히 발생한다. 특히, 요즘의 고해상도 모니터를 사용하는 경우에는 최대로 높여 놓은 화면의 해상도에서 타이핑 오류는 눈으로 찾기 어려울 수 있다.

```
...
if( 10 == x ) {
    ...
}
```

위와 같이 사용하는 것도 오류를 회피할 수 있는 한가지 방법이지만 웬지 어색한 것도 사실이다. 이런 것 들을 코딩룰로 만들어서 관리하는 것도 좋은 생각이다. 코드 리뷰 시에 이런 부분만을 중점적으로 검토 할 수 있도록, 체크 리스트를 만들어서 확인하는 것도 필요하다. 어떤 컴파일러의 경우에는 조건문에 할당문이 들어간 경우에는 경고 메시지를 보여주기도 한다. 즉, "if()"에는 항상 참이나 거짓을 묻는 질문만이 있는지 확인한다.

[조건문에서 불리언(Boolean)값만 사용한다.]

C언어는 "0"이외의 값은 "참(True)"로 판단한다. "-1"이나 "1, 2"처럼 "0"이 아닌 값들은 참이 된다는 것이다. 이런 것으로 인해서 개발자들이 흔히 "0"과 "그 외의 값"으로 참과 거짓을 표현할 수 있다. 하지

만, 이것은 자칫 실수하기 쉬운 부분이기에(실제로는 거짓을 원했지만, 참이 되는 경우가 더 많다.), 그런 식으로 사용하지 않는 것이 좋다. 차라리 더 명확하게 표현해 주는 것을 선호한다.

```
if ( x ) {
    ...
} else {
    ...
}
```

위와 같이 "x"의 값에 의존하는 것은 좋은 코드가 아니다. "x"가 "0"인 값을 가질 경우에만, "else"이하의 블록이 실행될 것이기 때문이다. 좀더 명확하게 의도를 밝혀서 코딩한다면, 오류도 줄어들며 코드를 읽는 사람도 이해하기 쉬운 코드가 될 것이다.

```
/* 명확한 조건을 명시한다. 결과는 Boolean값이 될 것이다. */
if( x == 0 ) {
    ... /* 주의!!! 조건이 바뀌었으니, 실행하는 순서를 변경해 주어야 한다. */
} else {
    ... /* 주의!!! 조건이 바뀌었으니, 실행하는 순서를 변경해 주어야 한다. */
}
```

위와 같이 표현을 바꾸면 실행의 순서를 변경해야 할 필요가 있다. 물론, "else"대신에 "else if"와 같은 것이 더 추가되어 나온다면, 좀 더 자주 실행될 가능성이 높은 것을 "if()"와 가까운 곳에 두는 것이 좋다. 여기서는 부정적인 것보다는 긍정적인 "조건문"을 만들어 주기 위해서 순서를 변경했을 뿐이다. 긍정적인 조건문이 이해하기 더 편하기 때문이다.

```
if ( x == 0 ) {
    ... /* 즉, x가 0이 될 가능성이 높은 경우에 "if()"와 가까운 곳에 실행될 명령문들. */
} else if ( y != 0 ) {
    ... /* y가 0이 아닌 경우가 더 많다면, "if()"조건문에 가깝게 위치. */
} else if ( z > 0 ) {
    ... /* 마찬가지로, "z > 0"인 경우가 더 많다면, 조건문과 가까운 곳에 해당하는 명령문들. */
}
```

물론, 조건문이 위와 같이 여러 개가 이어져 있다면 코드를 읽는 사람은 힘들어 한다. 따라서, 위의 조건을 변경할 수 있다면 최대한 간단한 조건문으로 만들어주는 노력을 해야 할 것이다.

그리고, 될 수 있으면 "switch()"문은 사용하지 않는 것이 좋다. "switch()"문을 사용해야 하는 경우에 가능한 코드의 복제(Copy)를 줄일 수 있어야 하며, 각 "case"별 "break"와 "default"를 생략해선 안된다. 그리고, 각 "case"들은 간략히 구성하는 것이 "switch()"문을 한 눈에 파악할 수 있도록 해준다(읽기 쉬워진다.).

```
switch( input ) {
    case 'A':
        do_someting_A();
        break;
    case 'B':
        do_someting_B();
        break;
    ...
    default:
```

```

        do_nothing();
        break;
}

```

위와 같이 "switch()"문을 사용한다면, 될 수 있으면 모든 "case"에는 반드시 "break"가 있어야 한다, 만약, 연결된 "case"문이 같이 사용될 수 있으면, /* ... fall through ...(연속됨) */와 같이 반드시 코멘트로 표시해주는 것이 좋다. 이런 것들이 지켜지지 않으면, 나중에 코드를 수정할 때 오류를 만들기 쉽다. 또한, "default"문은 반드시 사용해야 한다. "case"문과 다음 "case"문 사이에는 가능한 명령문이 적어야 하며, 전체 "switch()"문의 길이도 짧아야 한눈에 알아보기 쉽다.

```

switch( model ) {
    case MODEL_A:
        do_something_model_A();
        break;
    case MODEL_B:
        do_something_model_B();
        /* Fall Through */
    ...
    default:
        do_nothing();
        break;
}

```

```

void (*do_something)( void );
do_something = do_something_model_A;
or
do_something = do_something_model_B;
...

if (!do_something()) {
    do_nothing();
}

```

만약, 유사한 형태의 "switch()"문이 프로그램의 여기 저기에 사용되고 있다면, 이는 "복제된 코드"의 문제를 유발할 가능성이 크다. 특히, 특정 모델이나 칩과 같은 것의 의존하는 기능을 구현하는 경우에 이런 경향이 발생하는데, 이때는 "함수 포인터"와 같은 것을 이용해서 인터페이스를 제공하는 방향으로 코드를 변경해야 한다. 그리고, 사용된 함수 포인터들에 대해서는 초기화 시 한 번만 실행되도록 만들어줄 필요가 있다. 기본적으로 특정 모델이나 칩에 의존적인 코드는 따로 분리된 디렉토리에서 관리하는 것이 확장을 위해서 더 좋다.

[“switch()”을 제대로 사용하기]

"switch()"문은 상수로 주어질 수 있는 조건에 대해, 각각의 경우를 나누어서 처리할 수 있는 제어 흐름을 제공해준다. 조건으로 주어지는 값이 상수와 같이 다를 수 있는 것이라면, 그 각각에 대해서 어떻게 처리해 줄지를 정해줄 수 있다는 점에서 다양한 조건을 한번에 효과적으로 처리할 수 있는 방법이다. 하지만, "switch()"문은 오용되는 경우가 많으며, 특히 확장이나 변경에 대해서 취약성을 많이 드러내는 부분 중에 하나다. 따라서, 될 수 있으면 "switch()"문의 사용을 줄이는 것이 유지보수에 유리하다.

```
switch( value ) {
```

```

case xxx :
...
    break;
case yyy : /* Fall Through */
case zzz :
case kkk :
...
    break; /* 아래와 같은 코드가 아니지만, 후처리가 필요한 경우 생략될 수 있다. */
case mmm :
...
    break;
default :
...
}

```

위와 같은 코드는 일반적으로 "switch()"문을 사용할 때 많이 볼 수 있다. "switch()"문의 문제는 크게 3 가지로 나누어 볼 수 있다. 첫 번째는 확장 시에 "case"를 추가하는 방법으로 코드를 만드는 경우다. 이와 같은 방법의 대표적인 경우가 새로운 제품의 모델을 지원해야 하는 경우다. 그 때마다, 이런 비슷한 종류의 코드들이 특정 모델에 대한 처리를 위해서 소스코드 전체로 흩어져 존재하게 된다. 이 문제는 "switch()"문을 한 곳으로 모으고, 각각의 모델에 대한 인터페이스를 분리된 함수로 만들어서 함수 포인터로 대체하는 방법으로 변경할 수 있다.

두 번째는 "case"문이 계속 늘어나는 경우다. 실제로 실무자들이 만들었던 코드에는 256개의 "case"를 가지고 있는 경우도 있었다. 이때는 더 좋은 방법으로 차라리 함수 포인터의 배열을 사용하는 것이 효과적이다. 늘어나는 조건보다는 인덱스 방식으로 함수 포인터의 배열을 접근하는 것이 더 효율이 높다. 나중에 새로운 조건값을 추가하는 것도 손쉬우며, 코드의 변경도 최소화 될 것이다.

세 번째는 "break"를 사용하지 않고, 다음 "case"문으로 진행하는 경우다. 흔하게 발생하는 실수지만, 잘 해결되지 않고 괴롭히는 문제가 될 수 있다. 위의 예처럼, "yyy", "zzz", "kkk"에 대한 처리 후에 "mmm" 케이스에 대한 처리를 연결해 놓았다. 물론, 이렇게 처리해야 하는 경우도 있을 것이다. 그렇다고 하더라도, "kkk"에 대해서 "break"를 가지는 것이 좋다. 그리고, 연결해서 처리해야 할 부분이 있다면, 그 부분은 함수로 만들어서 두 곳에서 각각 호출해 주면 된다. 이런 식으로 처리한다면, 코드의 중복이 문제될 것은 없다.

마지막으로 언급하고 싶은 것은, 각각의 "case"에 대해서 실행해야하는 코드가 긴 부분이다. 이때는 각각의 "case"별로 간격이 늘어나게 되어 코드를 이해하기도 어렵다. 이를 해결하는 방법은 간단히 함수를 만들어서 "case"에서 실행해야 할 코드를 따로 분리하는 것이다. 그렇게 만들면 코드를 좀더 효과적으로 이해할 수 있게 되고, 각각의 함수도 재활용 될 가능성이 높다. "switch()" 문의 코드 블록이 늘어나면 전체적으로 코드가 길어져 이해하기 쉽지 않다. 그리고, 가능하면 "default"문도 반드시 두어서 어떤 입력에 대해서도 처리된다는 것을 보장해야 한다.

["continue", "goto", "break"의 사용 줄이기]

루프(Loop)에서 제어를 갑자기 건너뛰는(이동하는) 것은 코드를 이해하기 어렵게 만들 수 있다. 따라서, 가능한 "continue"와 같이 프로그램의 실행 경로를 급하게 옮기는 명령문의 사용은 줄여야 한다. 대부분의 경우 "continue"는 추가적인 "if()"의 사용으로 대체할 수 있다. 만약, 구현이 복잡해 지지않는다면 (즉, 코드의 이해를 방해하지 않고, 코드가 길어지지 않는다면), 대체해서 다른 코드로 변경하는 것이 좋다.

```

for( int i; i < MAX_LENGTH; i++ ) {
    if( ... ) {

```

```

        continue;
    }
    do_something();
}

```

.....

```

for ( int i; i < MAX_LENGTH; i++ ) {
    if( !( ... ) {
        do_something();
    }
}

```

위의 코드에서 보듯이, "continue"를 쓰지 않더라도 자연스럽게 다음번 루프로 진행할 수 있다. 물론, 조건문인 "if()"에서 반대 조건을 따져봐야 한다는 점은 있지만, 이것도 이해를 쉽게 만들려면 "드모르강의 법칙"과 같은 것을 사용해서 부정이 아닌 긍정조건으로 변경할 수 있을 것이다. 혹은, 조건판단 부분을 "isXXX()"와 같은 간단한 "inline"함수로도 변경할 수 있다.

"for()" 루프가 두 번 연달아 있거나, 혹은 "for()"루프 내에 중첩된 다른 루프가 있을 경우에는 "continue"가 어떤 루프를 탈출하는지 모호할 가능성도 있다. 따라서, 될 수 있으면 제어가 급하게 움직이는 경우는 사용하지 않는 것이 코드의 이해에 도움이 될 것이다. 마찬가지로, "goto"와 같은 것은 될 수 있으면 사용하지 않아야 할 것이다. 물론, 사용해야 할 경우도 있지만, 그럴 경우는 코드의 이해를 떨어뜨리지 않는 단순한 곳으로 한정하는 것이 좋다. 복잡한 코드 내에서는 사용하지 않는 것이 원칙이다.

```

for ( int i = 0; i < MAX_LENGTH ; i++ ) {
    ...
    if ( array[ i ] == xxx ) {
        break;
    }
    ...
}

```

"break"를 사용하는 것도 루프에서 급하게 제어를 옮기기 때문에 될 수 있으면 줄여야 한다. 물론, "switch()"문의 경우는 예외적으로 "break"가 반드시 필요한 경우다. 그리고, "switch()"문은 조건문에 해당하지 루프에 해당하지도 않는다. 루프 내에도 "switch()"를 사용할 수 있지만, 문제될 것은 없다고 생각된다. 만약, "break"를 사용하지 않는다면, 어떻게 루프에서 빨리 빠져나올 수 있을까? 특히, 배열에서 어떤 원소를 찾거나, 혹은 어떤 조건을 만족할 때 루프를 바로 빠져나오고 싶을 때 어떻게 할 수 있을까? 이때 처리할 수 있는 방법은 "bool"값을 이용해서, 루프 탈출 조건에 추가하는 방식을 사용할 수 있다.

```

for ( int i = 0; (i < MAX_LENGTH) && ( not_found ); i++ ) {
    ...
    if ( array[ i ] == xxx ) {
        not_found = false;
    }
    ...
}

```

물론, 이에 반대되는 의견도 있는 사실이다. 즉, "break"를 사용하는 것이 오히려 코드를 읽는데 도움이 된다는 의견이다. 따라서, "break"에 대해서 중립적인 의견으로 결론을 내리는 것이 좋다고 생각한

다. 만약, 코드가 더 복잡해 진다면 "break"를 써서 해결하고, 코드 자체가 이미 복잡하다면 리팩토링 후에 "break"를 사용할지 결정하는 것이다. 추가적으로 예에서는 "for()" 루프가 일반적인 형태를 벗어난 것도 주의해야 한다.

```
int function( void ) {
    ...
    if ( xxx == yyy ) return zzz;
    if ( kkk == mmm ) return jjj;
    ...
    return aaa;
}
```

추가적으로 "return"은 함수의 구현에서 반드시 한 번만 나와야 할까? 예전에는 구조적인 프로그램을 만들기 위해서, 함수의 출구를 하나만 가지도록 만드는 것이 일반적이었다. 하지만, 최근에는 자원 누수(Resource Leak)과 같은 일이 없다고 생각되는 경우에는 "return"을 여러 개 사용해도 상관없다. 즉, "return" 이후에는 아무 일도 함수 내에서 발생하지 않기 때문이다(코드 읽기가 끝남).

["float"의 사용에 대한 주의]

"float"는 부동 소수점을 나타내기 위해서 사용하는 자료형이다. 다양한 유용성이 있지만 유효 숫자의 범위와 관련된 것은 기억해야 한다. 그리고, ARM과 같은 CPU에서 코드의 크기가 문제가 될 경우에 조심해서 사용해야 한다. 즉, ARM CPU Core에서는 부동 소수점을 지원하는 것이 추가적인 소프트웨어 라이브러리(Library)를 통해서 가능한 경우가 있다.

예전의 경험으로 생각해 보면, 운영체제의 코드에서 부동 소수점을 사용하는 경우는 없다. 운영체제의 경우 연산 능력보다는 주로 하드웨어에 직접적인 접근을 요구하는 코드가 많으며, 이런 하드웨어에 직접적으로 접근하는 코드는 부동 소수점 연산이 필요하지 않기 때문이다. 대부분의 경우 양의 정수 값을 사용한다. 특히, 디바이스 드라이버를 구현하는 경우 레지스터(Register)에 대한 읽기 쓰기 값은 항상 양의 정수다.

소프트웨어를 이용해서 부동 소수점 연산을 에뮬레이션(Emulation)해야 한다면, 관련 라이브러리로 몇 백 Kbytes정도의 공간이 더 필요하다. 또한, 하드웨어가 아닌 소프트웨어 연산을 통해서 구현하기에, 복잡한 연산의 경우 성능에 악영향을 줄 가능성도 있다.

부동 소수점 연산이 필요하지만 결과값은 정수로 한정되어야 하는 경우, 변환 테이블을 이용하는 방법을 사용해서 연산 자체를 없앨 수도 있다. 물론, 이때는 입력으로 받아들일 수 있는 입력 값이 한정된 경우가 더 효과적일 것이다. 대체로 부동 소수점 연산은 성능에 좋지 않은 영향을 주기 때문에, 될 수 있으면 사용하지 않는 것이 좋다. 변수 선언만 하더라도 라이브러리가 컴파일리에 의해서 기본적으로 추가될 가능성도 있다.

[변수의 크기(Size)에 민감한 코드]

정수를 표현하는 "int" 타입은 CPU의 아키텍처(Architecture)에 의존적이다. 즉, 32bit를 지원하는 CPU는 32bit크기를 가지며, 64bit CPU에서는 64bit를 가진다. 만약, 다양한 CPU에서 동작해야 하는 코드를 만든다면, 변수의 크기에 민감한 값들은 따로 타입을 정의해서 관리하는 것이 일반적이다. 즉, 각각의 사용하고자 하는 타입의 크기를 CPU별로 설정할 수 있도록 헤더 파일에 정의를 하고, 이를 CPU에 맞게 포팅(Porting)해서 사용한다. 예를 들어, 32bit 양의 정수만을 가지는 변수에는 "uint32_t"와 같이 "#define"으로 선언하고, 해당하는 CPU에 적당한 값으로 정하도록 한다.

```
#ifndef __TYPE_H__
#define __TYPE_H__
```

```
#define uint32_t unsigned int /* 32bit CPU에서 양의 정수만을 사용하겠다. */
#define uint8_t unsigned char /* 8bit 양의 정수만을 사용하겠다. */

...
#endif
```

이렇게 정의하고나서 모든 변수의 타입을 이것에 맞춰서 정의한다. 구현 파일에서는 "type.h"를 "#include" 시켜주기만 하면 된다. 이렇게 모든 변수의 크기가 특정한 크기를 가져야 할 필요가 있는 코드들은 하드웨어에 의존적인 코드인 경우가 많다. 그리고, 실시간 운영체제(Real-Time Operating System)의 경우에는 자료형에 대해서 특히 민감한 부분들이 있기 때문에, 이런 식으로 코딩을 해 주어야 한다.

변수를 선언할 때, 흔히 메모리를 더 적게 쓰는 방법을 생각해서 변수에 최적의 크기를 가지는 타입으로 정의하는 경우가 많다. 하지만, 이렇게 해서 얻는 효과는 크지 않다. 예를 들어, 32bit CPU에서 함수의 인수를 32bit 보다 작게 선언할 경우, 어차피 내부적으로는 32bit 스택과 같은 것을 이용하거나, 혹은 레지스터를 이용해서 값이 전달되기에 32bit를 다 사용한다고 볼 수 있다. 또한, 복귀 값(Return Value)으로 32bit 보다 작은 값을 주는 경우, 32bit 크기로 변환하는 과정이 내부적으로 추가될 수 있다. 이럴 경우에도 역시 그냥 32bit 복귀 값을 사용하는 편이 더 빠른 코드를 만드는데 도움을 줄 수 있다.

위에서와 같이 특정 크기를 가지는 변수들을 명시할 경우에는 다른 기본 자료형들도 다 정의해서 사용하는 게 좋다. 그리고, 일관되게 모든 코드에서 다 같이 적용하는 것이 좋다. 번거롭기는 하지만 나중에 코드를 다른 CPU 아키텍처에 적용할 경우 컴파일만 하면 실행될 가능성이 높기 때문이다.

정수를 나타내는 "int"에 대해서 좀 더 자세히 보도록 하자. "int"는 음의 정수, 0, 양의 정수를 다 포함하며, CPU 아키텍처에 따라 그 크기가 달라질 수 있다. 그리고, "if()"와 같이 사용될 경우 포함하는 값에 따라 달리 해석될 수 있다.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    unsigned int bigger = 8;
    int smaller = -9;

    puts("This is an Integer Comparison Test Program!!!");

    if ((bigger + smaller) < 4) {
        puts("This is TRUE!!!\n");
        printf("The value : %d\n", bigger + smaller);
    } else {
        puts("This is FALSE!!!\n");
        printf("The value : %d\n", bigger + smaller);
    }
    return EXIT_SUCCESS;
}
```

코드를 보면, "8 + (-9)"를 계산해서 "4"보다 작은지를 이용해서 값을 화면에 출력한다. 실제로 계산 결과는 "-1"로 "4"보다 작은 값이지만, 화면에 출력되는 값은 "This is FALSE!!!"를 볼 수 있을 것이다. 이유는 "if()"와 같은 조건절에서는 부호가 있는 값들이 부호가 없는 값으로 인식되어 "-9"가 "(0xFF - 9)"

로 해석되기 때문이다. 따라서, "if()"와 같은 조건 절에서 정수 비교에는 부호가 있는 정수와 부호가 없는 정수를 혼용해서 사용하면 안된다.

정수의 크기에 민감한 자료형을 정의하기 위해서는 "stdint.h"를 이용하는 것이 좋다. 이 헤더 파일에는 각종 정수의 크기 및 부호 여부에 따라 사용할 수 있는 정수형이 제공되고 있다. "stdint.h"는 "C99" 표준에서 채택되었으니(대부분 제공한다), 사용하는 컴파일러가 지원하는지 확인할 필요가 있다. "GCC(GNU C/C++ Compiler)"는 이미 제공되고 있다.

[할당문과 비교문]

할당 문은 변수에 특정 값을 저장하는 목적으로 사용한다. 비교문은 두 개의 변수나 상수 값을 비교할 경우에 사용한다. C언어에서 할당 문에는 "="(값을 저장소에 넣기)를, 비교문에는 "=="(참과 거짓을 판단)를 사용해서 혼동을 주는 경우가 흔하다. 예를 들어, "x=y"와 "x==y"는 서로 다른 의미를 가지기에 실수로 잘못 사용할 경우 찾아내기 어려운 버그가 되곤한다.

```
if( x = y ) {
    printf("The value of x and y is same!!!\n");
} else {
    printf("The value of x and y is not same!!!\n");
}
```

위의 예제는 x와 y의 값이 같은지를 확인하기 위해서 사용했지만, 실제로는 x의 값에 y를 넣어 0이 아닌 값이라면 항상 "The value of x and y is same!!!"이라는 출력 결과를 얻게 된다. 즉, "=="를 사용해야 하는데, "="를 사용해서 생겨나는 문제다. 버그를 찾기하기 위해서 논리보다 시력이 더 필요한 경우다. 좀 더 명쾌하게 의미를 전달하기 위해서는 "if(x = y)"를 차라리 함수의 형태로 만들 수도 있다.

```
BOOL isEqual( x, y ) {
    if( x == y ) return TRUE;
    return FALSE;
}

if ( isEqual( x, y ) ) {
    printf("The value of x and y is same!!!\n");
} else {
    printf("The value of x and y is not same!!!\n");
}
```

물론, 이 경우에는 지나치게 추상화 시킨 경향이 있기는 하지만, 어쨌든 직접적으로 보기에 문제가 생길 여지가 줄어든다. 즉, 한번에 코드를 다 읽지 않고 두 부분으로 나누어서 이해해야 할 부분을 줄여주고 있기 때문이다. 이런 식으로 복잡한 조건(Condition)에 대한 처리를 분리된 함수로 만들 수 있다.

기본적으로 "if()"에는 조건을 판단할 수 있는 문장만 포함되어야 한다. 값을 이용하지 말고 "참인지 거짓 인지를 명확하게 판정하는 문장"을 가지고 있어야 할 것이다.

["const"의 사용]

"const"는 상수(constant)값을 가지도록 만드는데 사용하는 형 지정자이다. "const"로 선언된 변수는 값이 변경될 수 없기에 함수의 인자로 넘겨지는 변수들을 보호하기 위해서 사용할 수 있다. 될 수 있으면 헤더 파일과 같은 곳에서 값이 변경되지 않는 함수의 인자를 정의할 때 "const"를 사용하는 것이 좋다. 왜냐하면, 함수를 호출하는 측에서는 함수가 어떤 인자들의 값을 변경할지를 미리 예상할 수 있으며, 함수를 구현하는 측에서는 실수로 인자를 바꿀 위험이 적기 때문이다. 따라서, 생각지 못한 부수적인 효과(Side Effect)를 예방하는데 사용될 수 있다.

```
#include <stdio.h>
#include <stdlib.h>

unsigned int factorial(const unsigned int nth) {
    unsigned int result = 1;

    for (int i = 1; i <= nth; i++) {
        result = result * i;
    }
    return result;
}

int main(void) {
    puts("Factorial Demo");

    unsigned int nth = 10;
    printf("The factorial of %d : %ld\n", nth, factorial(nth));

    return EXIT_SUCCESS;
}
```

위의 코드는 팩토리얼(Factorial)값을 계산하는 간단한 코드이다. 여기서, "factorial()" 함수의 인자로 "nth"가 "const"로 선언되었다. 함수 내에서는 "nth"를 변경하려는 어떠한 코드도 컴파일 시에 오류를 발생 시킬 것이다. 따라서, 프로그래머의 실수로 인한 변경을 컴파일러 차원에서 방지해 준다.

매직 넘버(Magic Number)라고 부르는 코드 내의 각종 이름 붙여지지 않은 상수 값을 정의하는데도 "const"는 유용하다. 예를 들어, "3.14"라고 원주율을 그대로 코드에서 사용하는 것보다, 이를 변수를 정의해서 "const int PI=3.14"로 하는 것이 코드를 보는 입장에서는 훨씬 더 읽기 편하다. "3.14"는 그나마 "PI"라는 값을 이미 아는 사람이 해석하기 쉽지만, 코드에 이해하지 못하는 숫자들이 있는 경우 코드를 읽는 사람들에게 좌절을 주기 때문이다.

"const"로 선언될 수 있는 것들은 변경이 불가능한 것들이다. 이런 것들은 될 수 있으면 전부 "const"로 선언해 주어야 실수로 변경되는 것을 막아줄 수 있다. 이런 것들 중에서 특히 상수 값들은 "const" 변수를 사용해서 "이름(Name)"을 부여할 수 있기에, 단순히 "#define"과 같은 전처리 명령을 이용하는 것보다 디버깅 시에 더 많은 정보를 줄 수 있다. 따라서, 될 수 있으면 상수 정의를 "#define"보다 "const"를 이용하는 편이 좋다.

"const"를 이용해서 포인터를 선언하는 것도 가능하다. 이때는 다음과 같은 두 가지 경우가 존재할 수 있다. 즉, 타입을 먼저 두는 경우와 const를 먼저 두는 경우다.

```
init x = 100;
int const* pointer = &x;
const int* pointer = &x;
```

"int const*"와 "const int*"는 사실상 동일한 효과를 가진다. 즉, 변수 "pointer"가 가르키는 내용이 "const"이기에 "pointer"를 이용해서 변경하려고 할 때, 컴파일러는 오류를 발생시킬 것이다. 이미 "pointer"가 가르키는 부분이 변경이 불가하다는 정보를 알기에, 컴파일러는 "pointer"를 이용해서 접근하는 변경에 대해서 오류를 눈치챌 수 있다. 이와 같이 복잡한 것들을 만났을 때는 정의되는 오른쪽에서 왼쪽으로 읽어가는 방법으로 해석하면 된다. 즉, "pointer"는 "int const"를 가리키는 포인터가 된다.

“const”로 선언된 변수들은 선언과 함께 초기화를 해 주어야 한다. 예외적으로 함수의 경우에는 초기화 할 필요가 없다. 나중에 변경이 불가능하기에 미리 초기화를 해주는 것이, 컴파일러가 “const”를 만났을 때 변경하지 못하도록 만들어주기 때문이다. 위의 예에서 “pointer” 자체를 “const”로 만드는 것도 가능하다.

```
int const* const pointer = &x;
const int* const pointer = &x;
```

즉, 이것을 읽으면 다음과 같은 뜻이 될 것이다. “pointer”는 `const`이며, `const integer`를 가리키는 포인터(*)이다.“와 같은 뜻이 된다. 이때는 “pointer” 자체도 “const”가 되기 때문에, 다른 변수를 가르키는 포인터의 역할은 할 수 없으며, 단지 하나의 변수만을 가리키게 된다. 여러 번 인다이렉트(Indirect)한 방법으로 선언해줄수도 있지만, 복잡한 것보다는 간단하게 만들어서 사용하는 것이 버그를 줄이는데 유리하다. 읽기 힘든 코드는 될 수 있으면 만들지 않는 것이 좋으며, 즉각적으로 의미를 파악하기 힘든 코드는 만들지 않는 것이 원칙이다.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    puts("Const Demo");

    const char *constPointerforString =
        "This is a pointer for const string!!!\n";
    char const *pointerforConstString =
        "This is a pointer for string const!!!\n";

    char* const constPointer = "This is a const pointer!!!";

    printf("%s", constPointerforString);
    printf("%s", pointerforConstString);

    constPointerforString = pointerforConstString;

    printf("%s", constPointerforString);
    printf("%s", pointerforConstString);

    //constPointer = constPointerforString; /* bug!!! */

    printf("%s", constPointer);

    return EXIT_SUCCESS;
}
```

위의 예제는 변경이 불가능한 상수 문자열(Constant String)을 포인터를 이용해서 접근하는 것을 보여주고 있다. 즉, 포인터 자체는 “const”가 아니며, 포인터가 가르키는 것이 “const”다. 따라서, 포인터 자체는 변경이 가능하지만, 포인터가 가리키는 내용은 변경이 되지 않는다. 만약, 포인터 자체를 “const”형태로 만들고자 한다면, “`char* const pointer`”와 같이 선언되어야 한다. 이것은 “`char const* pointer`(혹은, `const char* pointer`)”와 다른 의미를 지닌다. 이런 경우에는 동적으로 새로운 것을 가리

키도록 포인터를 변경할 수 없으며, 정의할 때 초기화 해주어야 한다.

[“static”과 “const”的 사용]

“static”을 함수에 적용할 경우 정의된 함수가 파일 내에서만 유효한 범위를 가진다. 즉, 다른 파일에 정의된 코드들은 “static”으로 정의된 함수를 접근할 수 없다. “static”이 변수에 대해서 정의 되어도 마찬가지다. 즉, 파일 범위를 넘어서 다른 파일에 있는 코드에서는 해당 변수에 접근할 수 없게 된다. 예를 들어, “extern”으로 헤더 파일에 변수를 정의하더라도, 원래 정의된 파일에서 “static”으로 정의하면 컴파일러에서 오류를 발생시킬 것이다.

```
/* Variable.h */
#ifndef VARIABLE_H_
#define VARIABLE_H_

extern int global_variable;

#endif /* VARIABLE_H_ */

/* Variable.c */
int global_variable = 0; /* “static”으로 선언되면, 컴파일이 되지 않는다. */

/* StaticVariable.c */
#include <stdio.h>
#include <stdlib.h>

#include "Variable.h"

int main(void) {
    puts("Static Variable Example 01");

    global_variable = 100;
    printf("The value : %d\n", global_variable);
    return EXIT_SUCCESS;
}
```

따라서, 될 수 있으면 내부 구현 정보를 감추기 위해서 “static”을 적극적으로 사용하는 것을 고려하는 것이 좋을 것이다(정보를 감추는 효과 때문에). 물론, 함수 내부 구현에서 “static”을 사용하는 것은 변수의 값이 유지되는 것을 보장하기 위함이다. 함수 내에 “static”으로 정의된 변수들의 늘어나는 것은 함수의 재진입(Re-entrance) 문제를 발생시키기에 주의해서 한다. 즉, 멀티스레드 응용 프로그램을 개발할 때, 이전 상태의 기록이 유지되기 때문에 자원 공유에 문제를 발생시킬 수 있다.

```
/* message.h 파일 */
#ifndef MESSAGE_H_
#define MESSAGE_H_

#include <stdio.h>

static char* const constMessage = "This is a constant character string!!!\n";
void printStaticConstMessage(void);

#endif /* MESSAGE_H_ */
```

```
/* printStaticConstMessage.c 파일 */
#include "message.h"

void printStaticConstMessage(void) {
    printf("Called Function : ");
    printf("%s", constMessage);
}
```

```
/* Main 함수 파일 */
#include <stdio.h>
#include <stdlib.h>

#include "message.h"

int main(void) {
    puts("StaticConstDemo");

    printf("Main Function : ");
    printf("%s", constMessage);

    printStaticConstMessage();
    return EXIT_SUCCESS;
}
```

위의 코드는 "const"로 정의된 문자열 포인터를 각각의 분리된 코드 파일에서 사용할 수 있도록 만든 예제다. 이 예제에서 "static"을 준 것은 동일한 변수를 각각의 코드에서 사용할 수 있도록 전역적으로 만들어준 것이다. 만약 "static"이 없다면, 컴파일은 통과할 수 있을지 몰라도, 링크(Link) 과정에 같은 변수가 두 번 정의되어 있다는 오류가 나올 것이다. 따라서, 이처럼 변수를 공유하기 위해선 "static"을 사용해서 하나의 변수를 공유할 수 있도록 해야한다. 물론, 이와 같이 전역 변수를 사용하는 것은 문제를 유발할 가능성이 있다.

[매크로(Macro)는 간단한(?) 것만]

C언어에서 매크로는 특정 코드를 대체하는 방법으로 주로 많이 사용한다. 하지만, 매크로의 잘못된 사용으로 엉뚱한 결과가 나오는 경우가 많으며, 항상 사용에 주의해야 한다. 이를 회피하는 한 가지 방법은 매크로 대신 "inline"함수를 사용하는 것이 있으며, 간단한 표현에만 매크로를 정의하는 방법으로 한정할 수도 있다. 어쨌든 될 수 있으면 매크로의 사용을 줄이고, 다른 대체할 수 있는 방법을 찾는 것이 좋다.

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

매크로의 약점은 그냥 주어진 표현을 대체한다는 점이다. 문맥과 같은 것을 고려하지 않고, 일단 무조건 대체한다. 위의 같이 매크로를 정의했다면, 어떤 문제가 발생할까? 단순히 "a"와 "b"가 단일 변수라면 특별한 문제가 생기지 않을 수도 있다. 하지만, "a"나 "b"가 다른 표현식을 포함하고 있다면 이야기는 달라질 수도 있다.

```
max( a + b ? c : d, a - b );
```

```
(( a + b ? c : d > a - b )? a + b ? c : d : a - b );
```

위와 같은 상황이 발생한다면, "max()"라는 매크로의 출력 값이 제대로 되었다고 보장할 수 있을까? 아마도 이것을 제대로 파악하는데는 상당한 시간이 걸릴 것이다. 따라서, 간단한 매크로라도 잘못 사용될 가능성은 충분히 있는 것이다. 될 수 있으면 간단한 경우에만 매크로를 사용하고, 필요한 경우에는 매크로의 파라미터들의 주변에 "괄호"를 사용해서, 표현 명확하게 만들어 줄 필요가 있다.

```
#define max( a, b ) ((( a ) > ( b )) ? ( a ) : ( b ))
...
max( a + b ? c : d, a - b );
.....
((( a + b ? c : d ) > ( a - b )) ? ( a + b ? c : d ) : ( a - b ))
```

위와 같이 정의된 매크로는 읽는 사람이 비교적 쉽게 이해할 수 있으며, 버그가 발생해도 쉽게 찾을 수 있을 것이다(버그가 발생하지 않을 것이다). 따라서, 매크로를 사용할 때는 정확히 의도하는 바를 명쾌하게 보여주기 위해서, 반드시 매크로의 파라미터 주변과 매크로로 정의된 표현 자체를 좀 더 명확히 만들어 줄 필요가 있다. 물론, 처음에는 이런 부분에서 쉽게 실수를 저지르지 않을지도 모른다. 하지만, 시간이 흘러 코딩 한 사람이 바뀌고 사용이 늘어난다면, 원인을 찾기 어려운 버그로 남을 가능성이 있다. 컴파일러도 이런 부분에 대해서 특별한 경고 메시지를 보여주지 않을 것이다.

복잡한 매크로들은 절대 사용하지 않는 것이 바람직하다. 이런 것들은 "inline"함수의 대상이 된다. 이렇게 정의된 함수는 이해하기도 편하며, 함수의 코드들이 코드에 직접 매크로와 같이 들어갈 수 있기에, 실행 속도 측면에서도 일반 함수보다 유리하다. 주의할 점은 너무 잣은 호출이 필요할 경우에는 그냥 함수로 만드는 것이 더 좋다는 점이다. 즉, "inline"으로 들어가면 코드의 크기가 증가할 가능성이 있기 때문이다.

```
static inline int max( int a, int b ) {
    return ( a > b ) ? a : b;
}
```

위의 코드에서 보듯이 "inline"함수는 일반 함수처럼 정의되지만, "static inline"이라는 것이 붙는다. 따라서, 파일 외부로 노출되지는 않는다. 정의한 파일 내부에서만 한정적으로 사용될 있다. 자주 사용될 필요가 있는 것들은 차라리 함수로 분리해서 가지고 있는 편이 낫다. 이런 함수들을 묶어서, "math_util"과 같은 모듈을 만들 수도 있을 것이다. 현재 "inline"함수들은 "C99"표준부터 지원되고 있으며, 컴파일러마다 사용하기 위해서 옵션을 주어야 할 경우도 있다. 이는 사용하는 컴파일러의 옵션을 확인해 보란다.

```
static inline int max_f( int a, int b ) {
    return (a > b ) ? a : b;
}
...
#define max1( a, b ) (( a > b ) ? a : b)
#define max2( a, b ) ((( a ) > ( b )) ? ( a ) : ( b ))

int a = 100;
int b = 10;
int c = 1;
int d = 0;

printf("The maximum value : %d\n", max1( a, b ));
```

```
printf("The maximum value : %d\n", max1( a + b, a - b ));
printf("The maximum value : %d\n", max1( a + b ? c : d, a - b ));
printf("The maximum value : %d\n", (( a + b ? c : d ) > a - b ) ? a + b ? c : d : a - b );
printf("The maximum value : %d\n", max2( a + b ? c : d, a - b ));
printf("The maximum value : %d\n", max_f( a + b ? c : d, a - b ));
```

(실행 결과)

```
The maximum value : 100
The maximum value : 110
The maximum value : 1
The maximum value : 1
The maximum value : 90
The maximum value : 90
```

위의 코드는 앞에서 이야기한 것을 각각 구현해서 실행해 본 결과다. 실행 결과에서 알 수 있듯이, 마지막 4개의 결과에서 매크로를 부주의하게 사용한 경우에 올바른 결과값을 구하지 못한다. 매크로를 제대로 구현하거나, 인라인 함수로 만든 경우에는 두 가지 경우가 각각 같은 값을 준다는 것을 알 수 있다. 3번쨰와 4번쨰는 원하는 값을 찾아내지 못하고 있기에, 나중에 코드의 변경 시 문제가 될 가능성이 있다.

매크로는 코드의 복잡성을 관리하는 방법이지만, 잘못 사용될 경우에 치명적인 결과를 만들 수도 있기에 조심스럽게 사용해야 한다. 될 수 있으면 다른 방법으로 대체하는 것을 권장하지만, 사용해야 할 경우에는 짧은 표현을 대체하는 것에 한정하는 것이 좋다. 참고로, 매크로에 사용 되는 이름은 31자 이내로 제한하는 것이 좋다. 컴파일러에 따라 그 이상의 글자를 구별하지 못할 수도 있기 때문이다. 그럴 경우, 매크로들이 여러번 정의된 것처럼 보이게 된다. 코딩 규칙을 검사하는 툴에서 이런 부분을 오류로 검출해내기도 한다.

[매크로(Macro)의 활용]

C언어에서 매크로는 일종의 대체(Replacement)로 활용되고 있다. 즉, 논리적으로 한 덩어리의 코드를 끌어서 처리할 수 있도록 만들어준다. 특히, 간단한 수식이나 자료구조의 조작, 함수 등을 대체할 수 있는 수단으로 사용될 수 있다. 일반적인 수식이나 자료구조의 조작은 프로그래머가 사용할 수 있는 어휘를 늘리는 역할을 하며, 함수의 대체는 다형성(Polymorphism: 이름은 같지만 여러가지 다른 형태를 제공하기 위해)을 제공하기 위해서 사용할 수 있다.

```
#ifdef __DEBUG__
#define DEBUG_PRINT( fmt, args... )    printf( fmt, ##args )
#else
#define DEBUG_PRINT( fmt, args... )    /* Nothing */
#endif
```

위와 같은 코드는 개발 중인 소프트웨어에서 “printf()”를 활용해서 디버깅 메시지를 출력할 때 흔히 사용할 수 있는 방법이다. 즉, 디버깅 목적으로 컴파일 할 경우에는 메시지를 출력하지만, 배포(Release)를 목적으로 할 경우에 아무런 메시지도 출력하지 않도록 만든다. 이런 방법들이 중요한 이유는 개발자들은 흔히 배포 시에도 메시지를 출력하는 코드를 만들어서 시스템의 이상 동작을 알 수 있는 로그(Log)를 만들려고 하기 때문이다. 비슷한 이유로 위와 같은 매크로를 정의하지 않고 “printf()”와 같은 출력 함수를 바로 사용해서 디버깅과 배포 목적의 코드를 구분하지 않는 경우도 흔하기 때문이다.

```
#include <stdio.h>
#include <stdlib.h>

#define __DEBUG__ 1
```

```
#if ( __DEBUG__ == 1 ) /* #ifdef __DEBUG__ , #if defined( __DEBUG__ ) */
#define PRINT_WARNING( fmt, args... ) printf( fmt, ##args )
#define PRINT_ERROR( fmt, args... ) printf( fmt, ##args )
#define PRINT_DEBUG( fmt, args... ) printf( fmt, ##args )
#define PRINT_LOG( fmt, args... ) printf( fmt, ##args )
#define PRINT_INFO( fmt, args... ) printf( fmt, ##args )
#else
#define PRINT_WARNING( fmt, args... )
#define PRINT_ERROR( fmt, args... )
#define PRINT_DEBUG( fmt, args... )
#define PRINT_LOG( fmt, args... )
#define PRINT_INFO( fmt, args... )
#endif

int main(void) {
    PRINT_DEBUG("This is a Test Message!!!\n");
    return EXIT_SUCCESS;
}
```

디버깅을 목적으로 한다면, 디버깅 목적을 세분화 해야한다. 크게 "디버그(Debug), 경고(Warning), 오류(Error), 로그(Log), 정보(Information)"로 나누어서 각각에 맞는 매크로를 정의해서 사용하는 것이 좋다. 추가적으로 개별 모듈별로도 메시지 출력력을 끄고 켤 수 있는 매크로를 따로 가져가는 것이 좋다. 디버그 시나 기타 발생하는 메시지를 자신이 개발하고 있는 모듈에 한정적으로만 적용하고, 다른 사람이 개발중인 모듈에서 발생하는 메시지와 섞이지 않도록 만드는게 효율적이기 때문이다.

```
#define __MODULE_PRINT__ 0
#define __MODULE_SEND__ 0
#define __MODULE_MYMODULE__ 1
...
```

위와 같이 정의한 후에, 각각의 모듈별로 디버그 매크로를 정의한 파일을 만들어서, 앞에서 정의한 모듈별 메시지의 끄고 켜는 것을 정의한 헤더 파일을 include하면 될 것이다. 모듈별로 자신만의 매크로를 일관되게 사용할 수 있다면, 실제 실행에서 발생하는 디버깅 메시지의 양도 적으며 찾기도 편할 것이다. 또한, 수준(Level)별로 메시지의 출력을 정하면, 정말 원하는 메시지만 골라서 볼 수도 있다. 이런 것들은 주로 과제의 초반에 개발자들간의 협의로 정해고 과제 완료까지 일관되게 적용하는 것이 좋다.

매크로를 사용할 때는 주의해야 한다. 잘못된 대치(Replacement)로 인해서 발생하는 오류들은 찾기 어려우며, 매크로 정의가 사용되는 곳을 제한하기도 어렵다. 즉, 파일 전체적으로 영향을 줄 수 있기 때문이다. 만약, 주요 헤더 파일에 정의된 매크로라면 그 영향력이 더 커진다. 원칙적으로 전역 자료 구조나 전역적으로 사용하는 함수를 가져가는 것은 옳지 않은 일이다(물론, 예외도 있다). 매크로로 길게 코드를 작성할 일이 있다면, 차라리 함수로 선언해서 사용하는 것이 좋다. "inline" 함수의 경우가 대표적인 예이다. 매크로로 간단한 식을 만들 때도 "inline" 함수를 활용하는 편이 좋다. 매크로로 상수를 정의하는 것도 상수 변수를 정의하는 것으로 대체할 수 있다.

```
#include <stdbool.h>
#include "stack.h"

static inline bool isEmpty( STACK stack ) {
    if ( stack->top == stack->bottom ) {
```

```

        return true;
    }
    return false;
}

```

위의 함수는 간단한 “stack”의 빈상태를 검사하는 코드를 구현한 것이다. “stack”의 “top”과 “bottom”이 같을 때 “stack”이 비었다고 가정했다. 항상 “bottom”은 고정 상태로 두고, “top”을 움직인다고 보았다. 이 함수를 이용할 때는 단지 일반적인 함수와 같이 사용하면 된다. 매크로와 다른 점은 함수가 컴파일러의 의해서 매크로처럼 코드에 들어갈 것인지, 아니면 일반적으로 함수로 호출이 될 것인지가 결정된다는 점이다. 이와 같은 방법으로 매크로보다 읽기 쉬운 코드를 만들 수 있으며, 잘못될 확률도 줄어든다. 컴파일러가 알아서 처리할 것이기 때문에 매크로와 같은 단순한 대체(Replacement)와 다르다.

[매크로 사용 줄이기]

매크로(Macro)를 사용해서 함수와 같은 코드를 대체하는 것은 코딩에서 빼도록 하자. 흔히들 많이 사용하는 절대값(Absolute Value)이나 최소/최대 값과 같은 것을 찾는 매크로를 사용하고 있지만, 조금만 잘 못 사용해도 문제가 발생할 가능성이 높다. 전처리(Pre-Processing)를 위한 명령(Directive)에 사용하는 "#define"과 같은 것은 의미가 있지만, 특정 문자열을 이용해서 코드를 대체하는 것은 코드에서 문제를 일으킬 수 있다.

매크로는 조금만 잘못 사용해도 부수적인 효과(Side Effect)의 원인이 되거나, 컴파일 오류들을 발생시킬 것이다(컴파일 오류는 그나마 좋은 편이지만). 또한, 컴파일 시 문자열이 코드로 확장되는 과정에서 코드 크기를 늘리고 중복된 코드를 만들어 낼 수 있다. 차라리 매크로로 이 정도를 해야할 필요가 있다면, 다른 방법을 찾는 것이 좋다. “Inline”함수가 그 한가지 방법이다.

C나 C++언어는 타입을 컴파일러가 검사하는 언어이다. 정적으로 컴파일할 때 모든 함수와 변수의 타입들이 검사를 받게 된다. 호환되지 않는 타입들에 대해서는 경고 메시지나 에러 메시지가 표시될 것이다. 이런 사소한 타입 검사에 대한 오류들은 발견 즉시 해결되어야 하며 해결하려는 노력이 어렵지도 않다. 호환되지 않는 타입을 사용한 오류는 사소하지만 중요한 것들이기에, 반드시 컴파일 옵션을 "Warning" 메시지가 모두 표시되도록 설정해 주어야 한다.

"경고" 메시지가 출력 된다면 제거하는 것이 소프트웨어 개발자의 기본 마인드다. C언어는 기본적으로 함수를 위주로 구성된 언어다. 따라서, 매크로를 대체하는 가장 간단한 방법은 함수를 만드는 것이다. 함수를 사용한다면 매크로로 인해서 확장될 부분의 코드를 함수로 구성해, 코드 확장으로 인한 공간을 줄일 수 있다. 물론, 이렇게 하는 것이 함수 호출을 포함하기에 오버헤드라고 생각할 수 있겠지만, CPU 시간의 극히 일부만을 차지하기에 무시할 수 있을 정도다. 따라서, 매크로들을 헤더 파일에 넣어서 관리하지 말고, 지역적으로 반드시 필요한 경우로 제한하는 것이 좋다.

"inline"으로 구성된 함수를 사용할 수 있는 것은 C99표준을 지원하는 컴파일러가 있을 경우다. 함수 자체를 작고 한가지 기능만 수행하도록 만드는 습관을 익힌다면 코드는 점점 안정적으로 변할 수 있다. 긴 함수는 많은 역할을 하기 위해서 여러 가지를 한꺼번에 처리하는 경우에 만들어진다. 즉, 이렇게 다양한 일을 처리하기 위해서는 입력 변수의 수도 많아지게 되며, 함수가 의존하고 있는(호출하고 있는) 함수들도 점점 늘어나게 된다(의존성이 높아지게 된다.). 따라서, 당연히 이런 함수들은 테스트 하기도 쉽지 않으며, 버그가 생겼을 때 전체 함수를 다시 읽어서 이해해야 하기에 디버그하는 시간도 길어지게 된다. 논리적으로 분리된 이해하기 쉬운 크기의 함수들로 나눈다면, 문제는 지역적으로 한정된 부분에서 발생하게 되며, 각각의 문제를 해결하는 시간도 짧아진다.

매크로들을 잘 활용하려면 함수의 입출력과 관련된 문제를 찾는 것에서 좋은 예를 볼 수 있다. 즉, "ASSERT()"와 같이 단위 테스트(Unit Test)에서 사용하는 매크로들이 좋은 예이다. 이것들은 제품의

코드에는 들어가지 않지만, 제품에 들어가야 할 코드가 제 기능을 하는지 알려주는 도구로 사용된다. 따라서, 코드를 대체하는 것이 아니라 코드를 지탱하는 버팀목의 구실을 하는 것이다.

매크로는 괄호에 대한 처리를 제대로 하지 못한다. 그냥 무조건 대체만 할 뿐이다. 그리고, 매크로로 전달되는 모든 파라미터를 "문자열(String)" 그대로 대체해 준다. 따라서, 만약 변수들이 동일한 것이 여러 번 나타날 경우, 부수적인 효과(Side-Effect)를 발생시킬 수 있다. 이것은 개발자가 의도한데로 코드가 제대로 동작하기 어렵게 만든다.

조건부 컴파일을 위해서 매크로를 사용하는 것도 코드를 읽기 어렵게 만든다. 코드를 읽을 때 조건부로 선택되는 실행되지 않는 코드("#if 0 ... #endif")들은 모두 제거하는 것이 좋다. 조금이라도 코드를 읽기 편하게 만들어주는 것이 목적이다. 그런 코드들을 만나본 경험이 있는 개발자라면, 컴파일 옵션을 정해주는 것도 힘들다는 것을 알 것이다. 그렇다면, 이런 코드들은 어떻게 분리할 것인가? 가장 쉬운 방법은 컴파일에 필요한 파일들로 분리하는 것이다. 즉, 컴파일시 추가해야 할 파일들을 개발자가 선택할 수 있는 스크립트를 제공하면, 옵션에 따른 선택이 필요한 코드를 별도의 파일로 분리(함수로 분리)할 수 있을 것이다.

["#define"의 대체 "enum"]

"#define"과 같은 매크로는 값을 전처리 단계에서 문자 수준으로 대체하기에, 될 수 있으면 사용하지 않는 것이 좋다. 그 대신에 사용할 수 있는 것이 "enum"으로 상수 값을 표현해주는 방법이 있다. 예를 들어, 아래와 같이 PI를 상수로 대체하고자 한다면, 이때는 상수 변수를 사용해야 할 것이다.

```
#define PI 3.1415923597
```

.....

```
const double PI = 3.1415923597
```

여러 개의 관련된 정수를 "#define"으로 선언했다면, "enum"이 대체할 수 있는 방법이 된다. 관련된 정수들을 한번에 묶어서 정의하는 것은, 정보를 묶어서 관리할 수 있다는 점에서 편리하다. 무의미한 여러 개의 값을 "enum"을 사용해서 묶어주는 것은 추상화 측면에서 좋지 않다. 또한, "enum"을 사용하는 경우에는 상수 값을 정의하는데 일관된 규칙이 필요하다.

```
#define NUMBER_OF_LIST 100
```

```
enum {
    NUMBER_OF_LIST= 100
};
```

```
unsigned int array[ NUMBER_OF_LIST ];
```

이와 같이 하는 이유는 배열을 선언할 때도 상수 변수로 정의된 것이 사용 불가능 한 경우에도 "enum"을 사용해서 배열을 정의할 수 있기 때문이다. 하지만, C99와 같은 곳에서는 상수 변수를 사용해서 배열을 선언할 수 있기에 동일한 효과를 보인다. 그렇지만, 상수 변수의 경우에는 "Read-Only Data" 부분의 기억공간을 차지 하기에 극단적인 메모리를 절약하기 위한 코드에서는 조심해서 사용해야 할 것이다.

[인라인(Inline) 함수의 사용]

"inline"함수는 함수를 매크로 정의와 같이 사용할 수 있는 방법이다. "inline"함수로 만들면 매크로에서 발생하는 문제를 해결할 수 있으며(즉, 문자들을 그대로 대치(Replace)해서 발생하는 문제들), 함수로 호출할지 아니면 매크로와 같이 대치를 사용할지를 컴파일러가 알아서 처리할 수 있다. 컴파일러는 호

출이 찾을 경우 일반 함수처럼 처리할 것이며, 그렇지 않을 경우 매크로와 같이 인라인 함수내의 코드로 호출을 대체할 것이다. 컴파일러의 선택을 강제화하기 위한 방법도 있으니, 각각의 컴파일러의 사용 방법을 참고해야 할 것이다. "inline"함수는 파일 범위(Scope)에서 의미를 가지기에 "static"과 함께 사용해야 한다.

```
#include <stdio.h>
#include <stdlib.h>

static inline int fibonacci(int nth) {
    if ((nth == 0) || (nth == 1)) {
        return 1;
    }
    return fibonacci(nth - 2) + fibonacci(nth - 1);
}

int main(void) {
    puts("Inline Function Demo");

    printf("The Fibonacci value of %dth : %d\n", 10, fibonacci(10));
    return EXIT_SUCCESS;
}
```

링킹(Linking)란 심볼(Symbol)의 주소를 결정하는 과정으로, 파일에서 제공하는 링크는 크게 2가지가 있다. 각각은 외부 연결(External Linkage)과 내부 연결(Internal Linkage)로 나뉜다. "inline"함수는 내부 연결만을 제공하며, 외부에서 보여주지 않도록 파일 범위(Scope)를 가져야 한다. 파일 범위로 사용을 한정하기 위해서 반드시 "static"을 "inline"에 덧붙여 사용한다. 그리고, "inline"함수는 C99 표준에 추가된 것으로 GCC로 컴파일 할 때는 "-std=c99"를 옵션으로 주어야 한다. "static inline"이나 "inline static"을 사용하는 것은 아무런 차이가 없다.

"inline"함수는 정의된 파일내에서만 사용되어야 한다는 점에서, 여러 파일에서 사용하기 위해서는 헤더 파일에 정의한 후에 "include"를 이용해도 된다. 헤더 파일에는 실행 가능한 코드가 들어가선 안된다는 유일한 예외가 "inline"함수이다. 따라서, 자주 사용되는 "inline"함수들은 헤더 파일에 정의하는 것이 좋다. 물론, 자주 사용하게 되면 코드의 크기가 증가할 가능성성이 있다는 점을 기억해야 한다.

```
#ifndef __FIBONACCI_H__
#define __FIBONACCI_H__

static inline int fibonacci(int nth) {
    if ((nth == 0) || (nth == 1)) {
        return 1;
    }
    return fibonacci(nth - 2) + fibonacci(nth - 1);
}

#endif /* End of __FIBONACCI_H__ */
```

위의 경우는 "fibonacci()"함수를 "fibonacci.h"파일에 정의해서 여러 구현 파일에서 "include"해서 사용할 수 있도록 정의한 것이다. 따라서, 이렇게 구현한 파일을 여러 파일에서 충돌이 없이 사용하기 위해서는 "#ifndef ~ #endif"와 같은 것을 사용해서 한 파일에서 한번만 정의될 수 있도록 만들어 주었다.

["assert()"의 활용]

"assert()"는 실행시 반드시 확인하고 넘어가야 할 부분에 대해서 사용할 수 있다. 주로 디버그(Debug) 목적으로 개발 중에 필요한 항목을 확인하고자 사용된다. 또한, 대응하지 못하는 예외 상황이 발생했을 때 처리하기 위해서도 사용할 수 있다. 사용 방법은 간단히 확인이 필요한 논리적인 판단을 할 수 있는 조건을 "assert()"를 호출할 때 인자로 넣어주면 된다. 추가적으로 만약 조건이 만족될 경우에 한해서 출력할 문자열을 지정할 수 있는데, "assert(condition && "string")"과 같이 사용한다. assert()는 디버그 시에 사용되기에 배포(Release)에서 빠지게 된다. 즉, "NDEBUG"이 설정된 경우에만 효력을 가진다. 이 값은 Eclipse에서 디버그 모드로 빌드 할 경우에 기본으로 주어지기 때문에, 별도로 정의할 필요는 없다.

```
#include <stdio.h>
#include <stdlib.h>

#include <assert.h>

int main(void) {
    puts("Assert Demo");

    char printed_data[17];

    int result = sprintf(printed_data, "Hello, World!!!\n");
    assert((result == 15) && "The String Length is not 16!!!");

    printf("The Length : %d\n", result);

    return EXIT_SUCCESS;
}
```

위의 코드는 "sprintf()" 함수가 제대로 동작하는지 확인하기 위해서 "assert()"를 사용한 경우다. "sprintf()" 함수는 문자열을 지정된 주소로 복사하는 역할을 하며, 제어 문자열을 추가해서 "printf()"와 같은 방법으로 사용할 수 있다. 결과가 화면에 표시되는 것이 아니라, 지정된 주소에 저장된다는 점이 다르다. 여기서는 "Hello, World!!!\n"을 "printed_data[]"에 복사하고, 복사된 문자열의 길이를 "sprintf()" 함수가 복귀 값으로 돌려준다. 이 복귀 값이 원하는 값이 맞는지를 "assert()"로 확인했다. 그리고, 추가적인 정보를 위해서 메시지를 넣어주었다.

"assert()"의 다른 사용 예로는 단위 테스트(Unit Test)에서의 활용이다. 물론, 단위 테스트 프레임워크(Framework)을 사용한다면, "assert()"보다 프레임워크에서 제공하는 다른 매크로(Macro)를 사용하겠지만, 기본적으로 "assert()"와 하는 역할은 동일하다. C언어의 단위 테스트 대상이 되는 "단위"는 "함수"이다. 따라서, 함수를 호출하고 나서 제대로 함수가 동작했다는 것을 검증하기 위해서, 예상 값과 복귀 값을 비교하는 방법으로 기본적인 단위 테스트가 가능하다.

[함수의 호출값 확인]

소프트웨어의 오류가 가장 많이 발생하는 곳은 연결(Interface)부분이다. 즉, 하나의 함수를 호출하는 것에 대한 의미를 모른다거나, 그 함수가 돌려주는 값을 제대로 보지 않는다는 것이다. 어떤 함수가 어떻게 동작할지를 미리 알 수 있다면, 우리는 그것에 대한 대책을 세울 수 있다. 어떤 동작을 할지를 아는 방법은 그 함수의 매뉴얼을 읽어야 한다. 만약, 매뉴얼이 없는 함수라면 코드를 봐야 할 것이다. 코드도 없고, 매뉴얼도 없다면 그 함수의 사용 방법을 다른데서 찾아야 한다. 그리고, 대략적인 함수의 이해가 되었다면, 간단히 테스트할 수 있는 방법을 사용해서 코딩한다.

```
char *stringBuffer[ 100 ] = {0};
```

```
int result = sprintf(stringBuffer, "Hello, World");
assert(result == 13) && "sprintf() function works incorrectly!!!");
```

위의 코드를 잠시 보면, 함수를 하나 실행하고, 그 함수가 돌려주는 값을 "assert()"로 확인했다. 이런 것들이 모든 제공되는 함수에 대해서 필요한 것은 아니지만, 여기서는 간단히 어떻게 함수의 복귀 값을 확인 할 것인가 만을 보여준다. 즉, 모든 사용되는 함수 들에 대해서는 결과값의 확인이라는 과정이 추가되어야 한다. 물론, 복귀 값을 돌려주지 않는 함수 들도 있을 것이다. 중요한 것은 함수의 호출 이후에는 반드시 함수가 원하는 일을 처리 했는지 확인 하라는 것이다.

예를 들어, "malloc()" 함수는 메모리를 할당하는 역할을 한다. 메모리라는 자원은 실행 중에 할당 받고자하면 구현하는 소프트웨어에 동적인 상황을 추가하는 것이다. 즉, 메모리 할당은 시스템의 안정에 영향을 줄 수 있다. 할당 할 수 있는 메모리가 남지 않은 경우도 있기에, 반드시 복귀 값이 할당된 메모리에 대해서 제대로 된 주소 값인지 확인해야 한다. 그렇지 않고, 할당 받은(받았다고 생각하는) 메모리 공간에 대한 접근을 하게되면, 당연히 소프트웨어는 잘못된 주소공간을 접근해서 실행을 멈출 것이다. 임베디드 환경에서는 메모리를 다루는데 특히 주의해야 한다. Linux를 사용하더라도 메모리를 무한정 할당 할 수는 없으며(대략 2GB), PC보다 더 적은 메모리를 가진 곳에서는 경쟁으로 인해서 메모리가 부족할 가능성이 있기 때문이다.

함수의 호출 값을 확인하는 것이 오버헤드로 생각한다면, 호출 값을 만들 이유가 없다. 호출 값 자체가 함수의 실행이 올바른지 확인하기 위한 것이다. 따라서, 반드시 호출 값이 있는 함수에 대해서는 확인하는 절차를 코드에 포함해야 한다.

[조건문은 적게 사용하라]

조건문은 제어 경로(Control Path)를 나누는 역할을 한다. 즉, 실행 경로가 바뀌기 때문에 코드를 읽는 프로그래머가 생각을 더 많이 하도록 만든다. 특히, 여러 개의 조건문이 함께 오는 것을 경계해야 한다. 특히, 내부에 중첩된 "If()"문을 가지는 경우 코드를 복잡하게 만들 가능성이 높다. "switch()"문과 같은 경우에는 다양한 "case"문을 가지는데, 주로 특정 값에 의존적으로 분기(branch)하도록 만든다. 만약 모델이나 특정 옵션(option)에 따라 실행을 달리해야 하는 경우가 발생하면, 이런 코드들은 전체 코드에 여러 번 등장할 가능성이 높다. 이런 것들은 중복(Duplication, Copy & Paste)을 발생시키기에 방지할 필요가 있다. 중복은 프로그램에서 버그의 주요 원인이며, 중복을 없애주는 방법을 고안해야 한다.

중복을 없애는 방법은 여러가지가 있지만, 리팩토링(Refactoring)과 같은 것을 참고하면 해답을 찾을 수 있을 것이다. 중복이 발생하는 코드를 묶어서 하나의 함수(매크로보다 항상 함수를 사용하는 것이 더 좋은 선택)로 만들어서 중복을 대체 할 수 있다. 조건문이 복잡한 판단 로직을 가지고 있는 경우에는, 그 자체를 하나의 함수로 만들어서 "isXXX()"와 같이 TRUE나 FALSE를 돌려주도록 만들 수 있다. 중첩된 조건문은 분리된 if()문들로 만들어 보는 것도 좋다. 비슷한 "switch()"문이 여러 부분에 등장한다면, "switch()"문을 없애기 위해 함수 포인터의 묶음으로 선언된 구조체를 인터페이스 용으로 사용할 수 있다.

이런 모든 방법을 통해서 얻는 효과는 코드를 "읽기 쉽고, 고치기 쉽게" 만드는 것이다. 같은 이유로 반복적인 수정이 필요한 코드는 개선이 필요하다는 것을 의미하기에, 코딩 시 이를 유념해서 대체할 수 있는 방법들을 찾아야 할 것이다.

"switch()"문을 대체하는 방법 중에서 흔히 사용되는 것은 특정 형(Type)에 따른 분기를 없애는 것이다. 즉, "case"에서 분기하는 것을 미리 한번만 생성하고, 이후에는 동일한 타입에 대해서 일관된 사용을 보장하는 방법이다. C++과 같은 객체지향 언어에서는 "클래스(Class)"를 상속받아서 이를 해결할 수 있지만, C언어에는 상속의 개념이 없기에 그와 유사한 구조체를 이용한 방법을 사용한다. 예를 들어, 특정 모델인지를 물어서 모델 별로 해주어야 할 일을 "switch()"문으로 구현했다고 가정하면, 이런 코드들은 전체 소스코드에 널려있을 가능성이 높다.

개발자들은 “switch()”문을 사용하거나, 조건부 컴파일 등을 이용해서 코드를 만들려고 할 것이다. 하지만, 이런 코드들이 점점 더 늘어날수록 고쳐야 할 부분들이 늘어나게 되며, 점차 복잡한 양상으로 번져간다. 이런 경우에는 초기에 한번만 함수 포인터를 가지는 모델 별 구조체를 만든 후, 나중에 이것만 사용해서 코딩을 하게 만든다. 이때, 함수 포인터는 일관성을 가지고 정의되기에, 그것을 사용하는 코드들은 변경이 없게 된다. 단지, 실제로 구조체를 구현하는 함수만이 모델 별로 달라질 뿐이다.

```
typedef struct Model_Type {
    char *model_name;
    void (*handlerA)( struct ModelType *mt );
    void (*handlerB)( struct ModelType *mt );
    ...
} MODEL_TYPE;
```

위와 같이 정의된 자료구조를 프로그램의 초기화 시에 한번 생성하고 나면, “switch()”문에서 일일이 모델의 타입에 따라 분기해야 할 일이 줄어들게 된다. 위에서 보여준 “handlerA()”나 “handlerB()”는 실제로 타입에 따라 분기해서 해야할 일을 나타내는 함수에 대한 포인터이다.

```
switch( type ) {
    case TYPE_X :
        do_something( &model );
        break;
    case TYPE_Y
        do_something_more( &model );
        break;
    default :
        ...
}
```

위와 같이 정의된 “switch()”문은 “type” 변수에 의해서 분기하게 되며, 여기서 “type”은 모델의 타입을 알려주기 위해서 사용한다. 이를 대체하면 아래와 같이 될 것이다. 이때, “do_something()”과 “do_something_more()”는 각각의 모델 타입에 별도로 정의된 함수라고 가정한다.

```
MODEL_TYPE a = { "A Model", do_something };
MODEL_TYPE b = { "B Model", do_something_more };

...
a->handlerA( &a );
or
b->handlerA( &b );
```

물론, 여기서 보는 것은 간단한 예이다. 위의 코드에서 보듯이 한번 타입에 맞게 생성된 구조체를 이용해서 이후에 발생하는 모든 관련된 “switch()”문을 단순히 하나의 명령문으로 대체해 줄 수 있다. 나중에 추가적인 기능이 필요하다면, 구조체를 더 확장 혹은 추가하는 것으로 필요한 변경이 끝날 수 있다.

“if()”가 연속적이거나 조건문이 복잡하면 이해하기 어려운 코드가 된다. 이해하기 어렵다는 말은 버그가 그만큼 많다는 말과 같다. 이런 경우라면 “if()”문을 좀더 쉽게 만들어 주어야 한다.

```
if(( x >= 10 )||(y < 10) || ( z == 0 )) {
    ...
}
```

```
}
```

만약, 위와 같은 “if()”문이 있다면, 좀더 간단히 표현하기 위해서 다음과 같이 만들어 줄 수 있다. 여기서의 중요한 점은 읽기 쉬운코드를 만들어주는 것이다.

```
static inline int isOK( int x, int y, int z ) {
    return (( x >= 10) || ( y < 10 ) || ( z == 0 ));
}
```

```
...
if( isOK(x, y, z)) {
    ...
}
```

물론, 위의 코드보다 라인수는 늘어났다. 하지만, “inline”으로 선언된 함수의 이름을 제대로 사용한다면, 무슨 일을 하는 지는 함수 이름만 가지고도 알 수 있다. 또한, “inline”으로 선언된 함수를 다른 곳에서 재활용할 가능성이 있으며, 기존의 코드도 좀 더 이해하기 쉽게 바뀌었다. 이런 것들이 별로 도움은 안되고 코드의 길이만 늘리고 함수 호출 오버헤드(Overhead)가 있다고 생각할지도 모르지만, 그건 그렇게 보기 이기 때문에 그런 것이고, 실제로는 컴파일러가 효과적으로 처리해 주는 것들이다. 따라서, 이해하기 쉬운 코드를 만드는 것이 첫 번째 목표이어야 한다.

```
#include <stdio.h>
#include <stdlib.h>

enum MODEL_TYPE {
    A_MODEL = 0, B_MODEL, C_MODEL,
};

typedef struct _MODEL_TYPE {
    int code;
    char *name;
    int (*handler)(struct _MODEL_TYPE *model);
} MODEL_TYPE;

int model_handler(MODEL_TYPE *model) {
    printf("The model code : %d\n", model->code);
    printf("The model name : %s\n", model->name);
    return 0;
}

MODEL_TYPE a = { A_MODEL, "A Model", model_handler };
MODEL_TYPE b = { B_MODEL, "B Model", model_handler };
MODEL_TYPE c = { C_MODEL, "C Model", model_handler };

MODEL_TYPE *createModel(int model_type) {
    switch (model_type) {
        case A_MODEL:
            return &a;
            break;
        case B_MODEL:
            return &b;
    }
}
```

```

        break;
    case C_MODEL:
        return &c;
        break;
    default:
        ;
    }
    return NULL;
}

int main(void) {
    MODEL_TYPE *model;
    puts("!!!This is Remove Switch() Statement Example !!!");

    model = createModel(A_MODEL);
    model->handler(model);
    model = createModel(B_MODEL);
    model->handler(model);
    model = createModel(C_MODEL);
    model->handler(model);

    return EXIT_SUCCESS;
}

```

위의 코드는 구조체를 정의해서 필요한 모델을 생성한 후, “switch()”문을 사용하지 않고 해당 모델의 핸들러 함수를 처리해 타입 별로 달라지는 연산을 구현한 예이다. 생성자는 해당 모델에 대한 초기화 시 `g` 한번만 호출되기에, 코드에서 한 곳에서만 호출되며, 나머지 코드에서는 이미 생성된 모델의 핸들러 함수를 호출하는 것으로 “switch()”문을 대체해서 구현할 수 있다. 즉, 코드 전반에 타입 별로 된 switch() 문을 사용하지 않고도, 모델 타입에 따라 다른 행동을 취할 수 있도록 만든 경우다.

모델에 따라 조건부 컴파일을 이용해서 실행 프로그램을 만드는 것은 시스템을 정적으로 만드는 경우에 사용할 수 있지만, 조건부 컴파일이 복잡해지면 코드를 읽는 것을 방해하는 요소로 작용한다. 만약, 이런 상황이라면 차라리 특정 모델에만 필요한 코드를 분리해서 하나의 파일로 만들고, 컴파일러에 적절한 파일을 사용하라고 알려주는 방식을 이용하는 것이 좋다. 이런 식으로 만든 코드는 조건부 컴파일 관련 명령들을 없애서 더 읽기 좋은 코드를 만드는 것과 동시에, 새로운 기능의 추가나 변경에 대해서 민감하지 않은 코드를 만들 수 있게 한다. 따라서, 버그의 발생 가능성도 낮출 수 있게 되는 것이다. 중요한 점은 변경의 영향을 줄이면서 가독성을 높이는 길을 찾는 것이 무엇인지 아는 것이다.

[조건문의 분기를 간단하게 만들기]

“if()”문과 같은 조건문들이 여러 개가 나오면 코드는 쉽게 난잡하게 보인다. 사실 이런 일은 흔히 발생하는데, 코드를 개발하는 동안 이런 저런 이유로 여러 가지를 추가하다가 발생하는 일이다. 근본적인 해결책은 완성되었다고 판단되는 코드를 다시 수정하는 작업을 진행하는 것이지만, 그렇게 하기 위해서는 결국 “리팩토링(Refactoring)”과 같은 기법을 알아야 한다.

```

if( xxx ) {
    if( yyy ) {
        ...
    } else if( zzz ) {
        ...
    } else {

```

```

    ...
}
...
} else {
...
}

```

위와 같이 만들어진 코드는 길이가 지속적으로 증가하는 경향을 보이며, 위에서의 조건문을 이해했다고 아래로 이어지는 구조도 아니기에 점점 더 난해한 코드가 된다. "}"를 적절히 사용하고, 들여쓰기(Indentation)를 사용했지만, 그것 만으로는 쉽게 이해하기 힘들다. 문제는 여기서만 그치는 것이 아니다. 즉, 코드가 실행되는 경로가 여러 개이기 때문에, 테스트 해야하는 케이스도 늘어난다. 아마 이렇게 만들어진 코드는 한가지 이상의 일을 할 가능성도 높을 것이다. 이런 코드를 개선하는 목표는 "읽기 쉬운 코드를 만드는 것"과 "짧은 코드를 만드는 것", 두 가지 방향으로 진행되어야 한다.

```

if( xxx && yyy ) {
    ...
}

```

```

if( xxx && yyy && !(zzz)) {
    ...
} else {
    ...
}
...

```

위의 코드는 앞에서 본 코드를 좀 더 알기 쉬운 구조로 변경한 것이다. 물론, 조건식이 복잡해 졌지만, 그 문제는 `inline` 함수와 같은 것을 사용해서 간단히 만들 수 있다. 앞에서 보인 예는 내부에 조건문이 들어가 있는 구조였지만, 위의 코드는 이것을 풀어서 평탄한 구조로 만든 경우다. 따라서, `if()`로 묶인 단위로만 이해하면 된다. 코드의 수정도 좀 더 용이한 구조가 되었다. 코드하는 일을 동일하지만, 이해하기는 앞에서 본 코드보다 더 쉬워졌다고 느낄 것이다.

한가지 염두에 두어야 할 사실은 조건과 그 조건의 영향을 받는 행동은 가능한 가까이에 있는 것이 좋다. 그것이 멀어질 수록 사람이 기억해야 할 것이 늘어나기 때문이다. 코드의 구조를 약간만 개선해도 나중에 코드를 읽는 사람은 좀 더 쉽게 코드를 수정할 수 있게되기에, 전체적인 비용(=시간)을 줄일 수 있는 방법이다.

[조건문을 평평하게(Flat) 만들기]

복잡한 조건문(Conditional Branch)은 간단하게 만들어서 관리되어야 한다. 조건문이 복잡해지면, 코드는 읽기 어려워지는 것이 보통이다. 조건문 내부에 새로운 조건문을 가지는 형태의 중첩(Nesting)도 코드를 이해하기 어렵게 만들기에 주의해야 할 부분이다. 이런 경우를 코드에서 만난다면, 복잡한 조건문은 분리된 내부 함수(Internal Function : "static" 키워드를 사용하는)로 만들어주는 것이 좋으며, 중첩은 풀어주어야 한다.

```

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

```

```

typedef enum {

```

```
SUNDAY = 0, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY
} DAY;
```

```
static inline bool isWeekDay(DAY day) {
    if ((day == MONDAY) || (day == TUESDAY) || (day == WEDNESDAY)
        || (day == THURSDAY) || (day == FRIDAY)) {
        return true;
    }
    return false;
}

static inline bool isWeekendDay(DAY day) {
    if ((day == SUNDAY) || (day == SATURDAY)) {
        return true;
    }
    return false;
}
int main(void) {
    DAY today = TUESDAY;
    /* Nested Conditional Branch */
    if (isWeekDay(today)) {
        printf("Today is weekday!!!\n");
    } else if (isWeekendDay(today)) {
        printf("Today is weekend!!!\n");
    }
    /* Flattened Conditional Branch */
    today = SUNDAY;

    if (isWeekDay(today)) {
        printf("Today is weekday!!!\n");
    }
    if (isWeekendDay(today)) {
        printf("Today is weekend!!!\n");
    }
    return EXIT_SUCCESS;
}
```

위의 코드는 복잡한 조건문을 분리해서 "inline"함수로 파일 범위를 가지도록 만들어 주었다. 두 단계로 나누어서 조건을 만족시키는지 검사하는 코드를 별도의 조건문으로 만들어, 코드를 좀 더 읽기 쉽도록 해준 경우다. 물론, 여기서 보여주는 예제는 아주 간단한 것으로 실제 코딩에서는 조건문과 뭉어서 처리되어야 할 부분을 별도의 내부 함수로 만들어주는 것도 가능하다. 이때 짧은 함수가 아닌 경우에는 "inline"함수로 만들어선 안될 것이다.

한 가지만 덧붙이면, C99표준(Eclipse에선 Compiler Option으로 "-std=c99"추가 필요)에서 추가된 "Boolean" 타입을 위해서 "stdbool.h"파일을 "#include"했다는 것과, 변수의 타입으로 "bool"을 사용할 수 있다는 점이다. 자신이 만들어서 사용하기 보다 이미 있는 추가된 타입을 사용하길 권장한다. 그리고, 만약 사용하고 있는 컴파일러가 이것을 지원하지 않는 경우에만 자신의 타입을 만들어서 사용하더라도, "BOOLEAN"과 같이 표현해서 사용자가 추가된 타입이라는 것이 명확하게 보이도록 만들어주는 것이 좋다("DAY"라는 새로운 타입을 정한 방법과 마찬가지로).

[“switch()”의 사용 줄이기]

종종 다양한 입력에 대해서 각각의 처리를 달리해주어야 하는 경우가 있다. 프로토콜과 같은 것을 구현하는 경우, 가장 빠르게 처리할 수 있는 방법은 무엇일까? 일반적으로 아무 생각 없이 구현한다면, "switch()"문을 이용해서 각각의 입력 값(프로토콜 번호)에 맞는 "case"로 분기하는 형태로 처리할 수 있다. 하지만, 만약 구현해야 하는 "case"가 늘어난다면, 이것은 효과적인 처리 방법이 아니다.

이보다는 뭔가 개선된 방법을 동원해야 할지도 모른다. 그리고, "switch()"에서 분기하기 위한 조건인 정수값으로 정해지지 않는 입력인 경우에는 어떻게 할 것인가? 따라서, 우린 이런 경우 새로운 방법을 찾아서 처리해주어야 한다. 이때 중요한 점은 확장성 및 효율성을 둘다 고려한 선택이다. 먼저 가장 간단한 배열을 이용해서 정수를 입력으로 받는 경우를 생각해 보자.

```
#include <stdio.h>

typedef int (*function)(int x);
function array[100];

int default_handler(int x) {
    printf("Default Handler Called!!!\n");
    return 0;
}

int main(void) {
    for (int i = 0; i < 100; i++) {
        array[i] = default_handler;
    }
    return 0;
}
```

위의 코드는 함수 포인터를 이용해서, 100가지의 입력에 대해서 기본적으로 처리할 함수(default handler)를 설정하는 것을 보여준다. 핸들러와 같은 것을 사용하는 경우 이처럼 기본 처리 함수를 미리 전체 범위에 대해서 설정하는 것이 기본 핸들러를 빠짐없이 구현하는데 유리하다. 나중에 개별적인 함수들을 각각의 인덱스에 맞게 설정하도록 한다.

[해쉬(Hash)를 이용한 “switch()”문의 대체]

해쉬는 정보를 빠르게 찾으려고 할 때 도움을 받을 수 있는 자료구조다. 즉, 입력 값을 이용해서 바로 원하는 정보를 찾고자 할 때 주로 사용한다. 예를 들어, 특정한 문자열을 입력으로 받았을 경우, 그 입력된 문자열에 해당하는 저장된 값을 이용하고자 할 때 사용 할 수 있다. 먼저, 배열과 같은 경우를 생각해보자.

```
int array[ 100 ] = { 0, 1, ..., 99 };
```

이 배열에서 원하는 값을 찾기 위해 같은 값을 가지는 배열의 원소를 찾기 위해서 처음부터 다 뒤지거나, 혹은 정렬을 통해서 찾으려는 배열의 원소를 정렬한 후에 찾아내는 방법 등을 사용할 수 있다. 만약, 배열의 크기가 더 증가하거나, 인덱스(Index)로 사용할 수 있는 값이 정수가 아니라면, 배열을 사용하는데 오버헤드(Overhead)가 발생할 수 밖에 없다.

만약, "string"과 같은 것이 주어졌을 때, 이 "string"이라는 것을 이용해서 즉각 찾아낼 수 있다면 훨씬 더 효과적으로 원하는 값을 찾아낼 수 있을 것이다. 예를 들어, "string"의 각 문자의 아스키(ASCII)코드 값을 더해서 100으로 나눈 값에 해당하는 배열의 주소에 값을 저장한다고 가정하면, 입력으로 받은 문자열의 아스키 코드 값 변환을 이용해서 배열에 대한 인덱스를 만들 수 있다. 만들어진 인덱스로 이용해

서 즉각 원하는 값을 찾아낼 수 있다면, 효과적으로 배열을 접근하는 방법이 될 수 있을 것이다.

이와 같은 방식으로 자료구조를 저장하고, 원하는 값을 추출하는 방식이 바로 해쉬(Hash)다. 물론, 해쉬를 사용하는 것에 대해 약점도 있을 수 있다. 즉, 키(Key)로 사용되는(인덱스를 만들기 위해서 사용되는) 값의 빈도가 높아지면, 찾기를 원하는 값이 달라도 같은 인덱스를 만들어 낼 가능성이 있으며, 이때는 둘 이상의 값을 구분하기 위한 추가적인 작업이 필요하게 된다. 하지만, 만약, 충돌이 잘 발생하지 않는 방법으로 인덱스를 만들어낼 수 있다면 문제를 일부분 회피할 수 있을 것이다. 추가적인 검색은 동일한 키 값에 대한 충돌이 발생하는 값들만 대상으로 할 수 있을 것이다.

해쉬를 이용해서 구현할 수 있는 것은 빠른 검색이 요구되는 분야이며, “switch()”문과 같은 것을 과다하게 사용하는 경우에도 대체 역할로 사용할 수 있다. 예를 들어, 복잡한 프로토콜(Protocol)을 구현할 경우, 각각의 특정 프로토콜 명령어(Command)에 대해서 함수의 포인터를 기억하고, 나중에 이 번호를 즉각 접근하는 경우도 생각할 수 있다. 특히, 프로토콜 명령어가 XML(Extensible Markup Language)인 경우, 문자열을 처리해야 할 경우도 해쉬가 도움이 될 것이다. 물론, 이때는 프로토콜 명령어를 해쉬에서 사용하는 키 값으로 변화하는 부분이 필요하지만, “switch()”를 사용하는 것보다 더 효율적으로 처리 속도를 개선할 가능성은 있다.

아래의 코드는 입력으로 주어지는 키 값을 이용해서 간단히 해쉬를 구현한 예제를 보여 준다. 해쉬 테이블을 생성하고, 새로운 엔트리를 추가하고, 찾기를 원하는 키 값을 통해서 해쉬 테이블에 이미 있는지를 확인하는 것 등을 포함하고 있다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct entry ENTRY;
struct entry {
    char *key;
    char *value;
    ENTRY *next;
};

#define MAX_HASH_SIZE 100
typedef struct hash_table HASH_TABLE;
struct hash_table {
    ENTRY *entry[MAX_HASH_SIZE];
};

static unsigned int makeHashIndex(const char *key) {
    unsigned int result = 0;

    char temp = *key;
    while (temp != '\0') {
        result += temp;
        temp = *(++key);
    }

    result = result % MAX_HASH_SIZE;
    return result;
}
```

```

static ENTRY *allocateHashEntry(const char *key, const char *value) {
    ENTRY *temp;

    if ((temp = (ENTRY *) malloc(sizeof(ENTRY))) == NULL) {
        printf("Cannot allocate hash entry!!!\n");
        return NULL;
    }

    if ((temp->key = (char *) malloc(strlen(key) + 1)) == NULL) {
        printf("Cannot allocate hash key!!!\n");
        free(temp);
        return NULL;
    }

    if ((temp->value = (char *) malloc(strlen(value) + 1)) == NULL) {
        printf("Cannot allocate hash data!!!\n");
        free(temp);
        free(temp->key);
        return NULL;
    }
    return temp;
}

static void copyHashEntry(ENTRY *entry, const char *key, const char *value) {
    strncpy(entry->key, key, strlen(key) + 1);
    strncpy(entry->value, value, strlen(value) + 1);
    entry->next = NULL;
}

static ENTRY *makeHashEntry(const char *key, const char *value) {
    ENTRY *temp = allocateHashEntry(key, value);

    if (temp == NULL) {
        printf("Cannot allocate hash entry!!!\n");
        return NULL;
    }
    copyHashEntry(temp, key, value);
    return temp;
}

static void insertEntry(HASH_TABLE *hashTable, const int index, ENTRY *entry) {
    ENTRY *temp = hashTable->entry[index];
    if (temp == NULL) {
        hashTable->entry[index] = entry;
        return;
    }
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = entry;
    return;
}

```

```

}

HASH_TABLE *createHashTable(void) {
    HASH_TABLE *temp;

    if ((temp = (HASH_TABLE *) malloc(sizeof(HASH_TABLE))) == NULL) {
        printf("Cannot create hash table!!!\n");
        return NULL;
    }
    memset(temp->entry, 0, MAX_HASH_SIZE * sizeof(ENTRY *));
    return temp;
}

int insertHashEntry(HASH_TABLE *hashTable, const char *key, const char *value) {
    unsigned int index;
    ENTRY *entry;
    if ((entry = makeHashEntry(key, value)) == NULL) {
        printf("Cannot make hash entry!!!\n");
        return -1;
    }
    index = makeHashIndex(entry->key);
    insertEntry(hashTable, index, entry);
    return index;
}

ENTRY *findHashEntry(HASH_TABLE *hash, const char *key) {
    unsigned int index = makeHashIndex(key);
    ENTRY *temp = hash->entry[index];
    while (temp != NULL) {
        temp = temp->next;
    }
    return hash->entry[index];
}

ENTRY *getHashEntry(HASH_TABLE *hash, const char *key) {
    unsigned int index = makeHashIndex(key);
    ENTRY *temp = hash->entry[index];

    if (temp == NULL) {
        return temp;
    }
    hash->entry[index] = temp->next;
    temp->next = NULL;
    return temp;
}

void freeHashEntry(ENTRY *entry) {
    if (entry == NULL)
        return;
    if (entry->key != NULL)
        free(entry->key);
    if (entry->value != NULL)

```

```

        free(entry->value);
        free(entry);
        return;
    }

int main(void) {
    puts("Hash Table Example 01");
    ENTRY *entry;
    HASH_TABLE *hash = createHashTable();

    insertHashEntry(hash, "name", "SH Kwon");
    insertHashEntry(hash, "name", "MJ Yoon");
    insertHashEntry(hash, "name", "JY Kwon");
    insertHashEntry(hash, "name", "JW Kwon");
    insertHashEntry(hash, "address", "Kangnam Gu Chungdam Dong");

    entry = findHashEntry(hash, "name");
    if (entry != NULL) {
        printf("The value : %s\n", entry->value);
    }

    entry = getHashEntry(hash, "name");
    while (entry != NULL) {
        printf("The value : %s\n", entry->value);
        freeHashEntry(entry);
        entry = getHashEntry(hash, "name");
    }

    entry = getHashEntry(hash, "address");
    if (entry != NULL) {
        printf("The value : %s\n", entry->value);
        freeHashEntry(entry);
    }
    return EXIT_SUCCESS;
}

```

["#pragma once"의 사용]

C언어에서는 "#include"를 같은 헤더 파일에 대해서 여러 번 할 경우, 동일한 타입에 대해서(혹은, 동일한 이름에 대한) 여러 번 정의 되었다는 컴파일러 오류를 만들 수 있다. 이를 없애기 위해서 아래와 같은 방법을 주로 사용한다.

```

#ifndef __XXX_H__
#define __XXX_H__
...
#endif

```

이와 동일한 역할은 하지만 표준이 아닌 "#pragma once"라는 것이 있다. GCC나 Microsoft의 C/C++ 컴파일러에서는 이러한 "#pragma once"가 통하지만, 다른 컴파일러들에 대해서도 사용될 수 있는지는 확인이 필요하다. 어쨌든, 위의 예와 같은 것을 좀 더 쉽게 할 수 있기에, "#pragma once"를 사

용하는 것도 고려해 보는 것이 좋다. 다양한 과제에서 사용할 것으로 예상되고, 다양한 컴파일러에서 컴파일 될 것이고 생각된다면, 전통적인 방법에 따라, 위의 예제를 그대로 이용해도 될 것이다.

```
/* myHeader.h */
#ifndef MYHEADER_H_    /* 이 부분이 #pragma once로 대체 됨 */
#define MYHEADER_H_
#endif /* MYHEADER_H_ */
#pragma once

extern void print_hello_world(void);

/* myHeader.c */
#include <stdio.h>
#include "myHeader.h"

void print_hello_world(void) {
    printf("Hello, World!!!\n");
    return;
}

/* main.c */
#include <stdio.h>
#include <stdlib.h>
#include "myHeader.h"

int main(void) {
    puts("Header File Include Example 01");
    print_hello_world();
    return EXIT_SUCCESS;
}
```

위의 예제에서 “#ifndef MYHEADER_H_ ~ #endif”가 “#pragma once”로 대체 되었다. 실제 코드에서는 주석으로 처리된 코드를 남겨놓지 않아야 하기에, “//”으로 처리된 부분은 삭제해 주어야 한다.

[추상화의 적용]

추상화(Abstraction)는 인간이 가장 잘 하는 것 중에 한 가지다. 흔히 특정 사람을 부를 때 "이름"을 사용한다. 그 사람 자체가 누군지를 묘사하는 것이 아니라, 그 사람을 대표하는 "이름"을 부르는 것이다. 마찬가지로 코드에서도 특정한 일을 수행하는 함수를 가리킬 때, 함수의 이름을 사용한다. 그 함수 내부에서 일어나는 것들을 대표할 수 있는 이름을 붙이는 것은 당연한 일이다. 즉, 구체적인 것을 모르고도 우리는 그 함수를 불러서 사용할 수 있게되는 것이다. 그 함수가 사용하는 자료구조가 어떤 형태인지는 모르지만, "그 함수를 호출해서 우리가 얻을 수 있는 결과는 어떤 것이다."라는 것을 가정할 수 있기 때문이다.

호출하는 코드를 클라이언트(Client)라고하고, 호출받는 코드를 서버(Server)라고 할 때, 클라이언트와 서버 간에는 모종의 약속이 있게 된다. 즉, 서버에서 제공되는 서비스를 사용하기 위해서 클라이언트가 어떤 행동을 해주어야 한다는 것이며, 그러한 서비스를 충족시킬 의무는 서버 측에 있다는 것이다. 일종의 계약(Contract)이라고도 볼 수 있는데, 우리가 함수를 만들 때도 이러한 계약 관계를 명확히 해 주어야 한다.

즉, 클라이언트 코드는 함수의 내부는 모르지만, 함수가 제공하는 인터페이스(Interface)를 호출함으로써 원하는 결과를 얻을 수 있어야 하며, 서버는 계약에 명시된대로 동작해 주어야 할 의무를 가진다. 따라서, 클라이언트 코드는 인터페이스에 대해서 의존적이게 되며, 서버는 인터페이스만 지키면 내부적인 구조는 언제든 변경할 수 있다. 역으로 생각하면 서버의 코드는 인터페이스를 변경해선 안되며, 계약을 완수할 의무를 가진다는 것이다. 이를 의존성 역전(Dependency Inversion)이라고도 부른다.

추상화의 관점에서 C언어는 파일 범위에서 한계(Limit)를 사용한다고 볼 수 있다. 즉, 자료구조를 구현 파일에 담아두고, 이를 같은 파일에 있는 함수들이 이용할 수 있다. 물론, 헤더 파일에 선언해서 다른 파일에 있는 함수들도 사용할 수 있지만, 헤더 파일은 될 수 있으면 최소한의 정보만 공개하는 형태로 만들어져야 한다.

만약, 내부적인 자료구조를 알 필요가 있는 다른 파일에 정의된 함수가 있다면, 오히려 그 함수를 나누어서 자료구조에 접근하는 함수를 자료구조가 정의된 파일로 옮겨주는 것이 좋다. 즉, 의존성을 분리하는 것이다. 이렇게 해서 얻는 장점은 코드가 변경이 용이한 구조가 된다는 점이다. 단점은 함수를 통해서만 자료구조에 접근하기에 접근 경로가 길어질 수 있다는 점이다. 하지만, 얻는 장점에 비해서 단점은 극히 미미하기에(혹은, 거의 없기에), 추상화를 기본으로 한 코딩이 더 효과적이라고 볼 수 있다.

예를 들어보도록 하자. 스택(Stack)과 큐(Queue)라는 예제는 항상 코드에서 많이 등장한다. 스택은 먼저 들어가면 나중에 나오는 자료구조를 표현할 때 많이 사용되며, 큐는 먼저 들어가면 먼저 나오는 자료구조에 많이 사용된다. "stack.h"와 "queue.h"를 아래와 같이 정의할 수 있다.

```
#ifndef _STACK_H_
#define _STACK_H_

typedef struct stackStructure *STACK;

STACK createStack( void );
Bool pushStack( STACK stack, x );
int popStack( STACK stack );

#endif
```

먼저, 코드에서 사용할 자료구조를 선언해 주는 것을 볼 수 있다("stackStructure"). 그리고, 이를 사용하기 위해서, 생성하는 함수("createStack()")와 스택에 "push"와 "pop"을 담당하는 함수를 선언했다. "pushStack()"와 "popStack()" 함수를 위해서, 생성 시에 자료구조에 대한 포인터를 얻고, 이를 함수의 인수(Parameter)로 넘겨주었다. 이 함수들을 호출하는 코드들은 "stack.h"파일을 "#include"시켜주어야 하겠지만, "stackStructure"가 어떻게 구현되었는지 알 필요는 없다. 즉, 자료구조에 대한 내부를 직접적으로 접근할 수 없게 된다.

[구조체와 함수의 결합]

객체지향 언어에서는 데이터와 그것을 다루는 함수를 한 곳에 묶어서 "객체(Object)"를 만든다. 그리고, 그 데이터에 대한 접근은 같은 객체 내에 선언된 함수로 제한한다. 이렇게 만드는 이유는 "정보의 숨김(Information Hiding)"을 통해서, 코드의 의존성(Dependency)를 낮추면서 응집력(Cohesion)을 높이기 위함이다. 즉, 데이터가 있는 곳에 함수를 같이 정의해서 데이터에 대한 임의적인 접근을 막아주고, 데이터와 관련된 함수들만 접근할 수 있도록 만든다. 객체들은 서로 통신하기 위한 창구(혹은 방법)로 객체가 제공하는 함수만 사용해야 하며, 직접 다른 객체의 내부 구조를 들여다 보지 못한다.

다른 말로 표현하면 내부 구조를 몰라도 상관이 없다는 뜻이며, 이는 나중에 외부에 제공되는 함수가 같은 형식이라면, 내부는 어떻게 변경되어도 상관없다는 뜻이다. 따라서, 독립적으로 객체에 대한 새로운

기능의 추가나 자료구조의 수정을 용이하게 만든다. 물론, 외부로 보여주는 함수가 인터페이스만 동일하고 하는 일이 달라도 된다는 뜻은 아니다. 만들어지고 사용되는 인터페이스들은 전부 "계약(Contract)"에 따라서 주어진 입력에 따라 예상되는 출력을 보여줄 의무가 있다.

C언어에서 제한적으로 이런 것들을 시도해 볼 수 있는데, 마치 하나의 생성된 객체처럼 구조체(structure)를 이용해서 자료 구조와 함수를 같이 정의하는 방법이다. 즉, 함수가 필요로 하는 자료 구조 자체에 대한 포인터를 두고, 함수의 인자로 자료 구조를 넘겨주는 방법이다. 물론, 상속이나 그런 개념은 없다. 하지만, 이 자료 구조를 사용하는 측에서는 정의된 인터페이스를 이용해서 자료 구조에 대한 조작을 할 수 있는 점과, 동일한 자료 구조라면 어떤 구현이 와도 일관성있게 사용할 수 있는 이점이 생긴다. 다음의 예를 보도록 하자.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct list {
    char *name;
    char **elems;
    void (*handler)(struct list *);
} list;

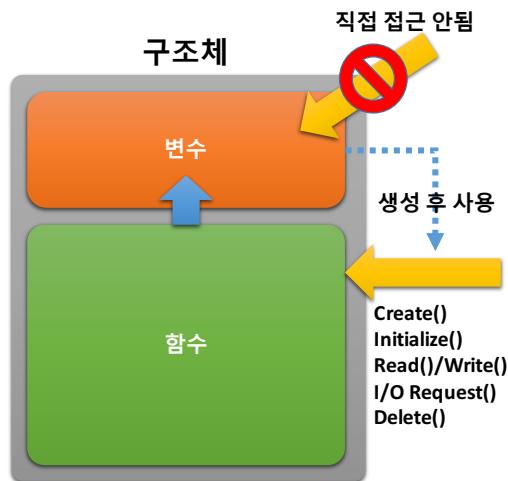
static void handle_print_list(struct list *list) {
    printf("Printing List\n");
    while (strcmp(*(list->elems), "END") != 0) {
        printf("List Elements : %s\n", *(list->elems));
        (list->elems)++;
    }
    return;
}

struct list mylist = { .name = "My List",
                      .elems = (char *[]) { "Books", "Notes", "Pencil", "Eraser", "END" },
                      .handler = handle_print_list };

int main(void) {
    mylist.handler(&mylist);

    return EXIT_SUCCESS;
}
```

여기서는 일반적으로 사용하는 "list"라는 구조체를 선언하고, 그 내부에 구조체를 다루는 함수에 대한 포인터(handler)를 두었다. 코드에서 간단히 특정 문자열들을 출력하는 것으로 만들었지만, 자료구조를 이용하는 다양한 함수들을 정의할 수 있다. 그리고, 해당 자료 구조를 이용해서 간단히 내부에 정의된 함수를 통해서 자료 구조 자체를 조작할 수 있다는 것을 보여준다. 만약, 동일한 인터페이스를 이용해서 다양한 기능을 제공하거나, 혹은 다양한 하드웨어를 다루어야 하는 등의 상황에서 이것을 응용해서 간단히 추상화(Abstract)을 구현해볼 수 있을 것이다.



예를 들어, 다양한 파일 시스템을 지원하려고 할 경우, 그것을 사용하는 측에서는 어떤 파일 시스템을 자신이 사용하는지 모르고도 사용할 수 있어야 한다. 이와 같은 추상화가 없이는 일일이 무슨 파일 시스템 인지를 확인해야 하고, 해당 파일 시스템에서 제공하는 함수를 이용해서 접근해야 할 것이다. 따라서, 코드가 해당 파일 시스템에 대한 의존성이 커지게 되며, 파일 시스템의 변경은 그것을 이용하는 코드의 변경도 동반할 수 있다.

하지만, 각각의 파일 시스템에 대한 접근이 단일한 인터페이스를 따른다면, 그것을 이용하는 사용자 측에서는 동일한 인터페이스를 사용해서 다양한 파일 시스템을 사용할 수 있게된다. 리눅스와 같은 커널에서도 이와 같은 것을 이용해서 파일 시스템과 장치 드라이버(Device Driver), 소켓(Socket) 등을 구현하고 있다.

[콜백(Callback) 함수의 구현]

콜백(Callback) 함수는 특정한 사건(Event)이 발생했을 때 호출해 달라고 미리 정의해 놓는 함수다. 따라서, 콜백 함수가 호출되는 상황은 호출되는 함수를 정의한 모듈의 컨텍스트(Context)가 아니라, 호출하는 곳의 컨텍스트가 연장된다고 생각할 수 있다(Unix와 같은 운영체제의 경우 시그널 핸들러(Signal Handler)는 커널에서 호출되지만, 실행 권한은 시그널 핸들러를 설정한 응용 프로그램으로 한정된다. 여기서는 운영체제 수준이 아니라 응용 프로그램 수준에서 콜백을 다룬다). 이런 콜백 함수는 주로 특정 비동기적인 이벤트(Event)가 발생했을 때 이를 처리하기 위한 경우가 많으며, 그런 이벤트의 결과를 즉각적으로 반영하기 위해서 사용된다. 콜백 함수로 구현할 경우에는 함수의 복귀 값이 의미를 가지는 경우는 드물기에(호출의 결과는 콜백 함수를 단순히 호출하는 측에서는 이용할 것이 없다), 주로 "void"를 사용해서 정의하는 경우가 많다.

```
#include <stdio.h>
#include <stdlib.h>

void (*registered_callback)(int ntimes);

void setCallback(void (*callback)(int ntimes)) {
    registered_callback = callback;
}

void callCallback() {
    (*registered_callback)(10);
}

void printHelloWorldNTimes(int ntimes) {
```

```

for (int i = 1; i <= ntimes; i++) {
    printf("<%2d>This is a HelloWorld print function callback!!!\n", i);
}
return;
}

int main(void) {
    puts("Callback Function Demo");

    setCallback(printHelloWorldNTimes);
    callCallback();
    return EXIT_SUCCESS;
}

```

위의 코드는 단순히 콜백 함수를 설치하는 것과 그것을 호출하는 것을 분리한 것으로, 실제 구현에서는 이벤트의 발생과 같은 경우에 호출할 수 있도록 변경해주면 된다. 한 가지 유념할 것은 콜백 함수를 사용한다고 해서 역 호출(Reverse Call)이라고 봐서는 안된다는 점이다. 역 호출은 상위의 모듈에 있는 함수를 하위의 모듈이 호출하는 것을 말하는 것으로, 이런 경우에는 모듈간의 결합도가 높아져 변경에 약한 코드가 만들어 진다. 즉, 호출은 상위에서 하위 모듈로 향하도록 만드는 것이 관리하기 좋은 코드를 만드는 방법이다.

콜백의 경우는 상위의 모듈이 하위의 모듈에 자신이 원하는 특정 이벤트가 발생할 경우 호출해 달라고 미리 호출될 함수를 설정한다. 따라서, 이 경우는 콜백 함수를 하위 모듈에 전달은 하지만, 하위 모듈의 입장에서는 단순히 자신이 가지고 있는 콜백 함수를 부르는 것 뿐이다. 상위의 모듈이 정의한 함수를 호출한다는 것은 알지 못하기에, 단순히 자신이 정의한 함수를 호출하는 것으로 생각하기 때문이다. 따라서, 이것은 역 호출 관계가 성립하는 것은 아니다. 또한, 하위 모듈은 상위 모듈에서 설정한 콜백 함수가 무엇인지 알지 못하며, 상위 모듈에서 설정한 콜백 함수가 달라지더라도 하위 모듈이 변경되지 않는다. 그 역도 마찬가지다.

[배열의 사용]

자료구조를 숨기는 것은 상세한 내부 구현을 상대로 부터 숨겨 구현의 자유도를 높이는 것이 핵심이다. 자료구조에 대한 직접적인 접근을 막고 간접적으로 접근 함수(Access Function)을 만드는 것은, 내부 자료구조의 변경에 대해 외부에 미치는 영향을 최소화 시켜준다. 외부에서 자료구조에 접근하고자 하는 코드는 반드시 접근 함수를 이용해야만 한다. 자료구조를 직접 접근하게 되면, 자료구조에 대한 변경이 발생했을 때, 해당 자료구조를 접근하는 모든 함수들은 변경에 대한 취약점을 노출하게 된다. 따라서, 될 수 있으면 자료구조는 한정된 범위에서 접근을 허용하는 것이 "관심의 분리(Separation of Concern)"라는 측면에서 좋은 디자인이라고 할 수 있다.

```

#define MAX_STACK_SIZE 10
typedef struct Stack {
    unsigned int top;
    unsigned int bottom;

    int stack[MAX_STACK_SIZE] = { 0 };

    bool (*pushStack)(struct Stack* stack, const int element);
    int (*popStack)(struct Stack* stack);
    void (*flushStack)(struct Stack* stack);
    unsigned int (*sizeofStack)(struct Stack* stack);
};

```

만약, 위와 같은 자료구조가 주어졌다고 하자. 여기서는 "struct"를 사용해서 구조체 정의를 이용했다. 자료구조 자체에 대한 정의 및 해당 자료구조에 접근하기 위한 함수들에 대한 포인터를 포함해서, 일종의 "Wrapper" 역할을 하는 자료구조를 정의했다. C언어에서는 "struct"내에 함수를 정의할 수 없기에 함수의 포인터들로 구성을 했지만, C++언어의 클래스와 유사하다는 것을 알 수 있다.

```
Stack stack;
```

```
stack.pushStack(&stack, 100);
int element = stack.popStack(&stack);
stack.flushStack(&stack);
```

내부에 정의된 배열을 외부에 노출시키지 않고, 외부에서는 정의된 자료구조를 이용해서 데이터를 스택에 저장하거나 가져올 수 있다. 만약, 외부 모듈이 배열이라는 것을 알게되면, 인덱스를 이용해서 직접 접근하려는 시도를 하게 될 것이고, 나중에 다른 구조로 자료구조를 변경하는 경우, 직접 접근하는 코드들은 전부 수정된 자료구조에 맞게 다시 작성해야 할 것이다.

[배열을 함수의 인자로 사용하기]

배열을 함수의 인자로 넘기는 경우, 일반적인 변수와 달리 주소로 넘어가게 된다. 만약, 주소가 아닌 복사된 값으로 넘어간다면, 배열의 크기를 고스란히 인자가 담아야 할 메모리 공간이 필요하며, 함수의 연산 결과도 원래의 배열에 영향을 주지 못하는 문제가 발생한다. 마치 포인터로 배열을 넘기는 것 같이, 배열이 인자로 사용될 때 배열의 주소를 그대로 가져오기에, 함수 내에서 배열을 조작할 경우 원래의 배열에 영향을 준다는 점을 주의해야 한다.

```
#include <stdio.h>
#include <stdlib.h>

void init_array_element(int array[]) {
    for (int i = 0; i < 100; i++) {
        array[i] = i;
    }
}

void change_array_element(int array[]) {
    for (int i = 0; i < 100; i++) {
        array[100 - (i + 1)] = i;
    }
}

void print_array_element(int array[]) {
    for (int i = 0; i < 100; i++) {
        printf("%3d", array[i]);
        if (i % 25 == 24) {
            printf("\n");
        }
    }
}

int main(void) {
    puts("!!!Array Passing Test!!!");
    int Array[100];
```

```

    init_array_element(Array);
    printf("====Before Changed====\n");
    print_array_element(Array);

    printf("====After Changed====\n");
    change_array_element(Array);
    print_array_element(Array);

    return EXIT_SUCCESS;
}

```

위의 코드는 100개의 정수로 구성된 배열의 원소를 1에서 100까지 각각을 초기화 한 후, 함수를 호출해서 원소들을 조작하는 연산을 실시했다. 조작된 결과를 확인하기 위해서 인자로 넘겨진 배열을 다시 1에서 100까지 출력해서 결과가 달라졌는지 확인하는 프로그램이다. 만약, 주소가 아닌 복사방식으로 함수 호출 시에 배열이 인자로 넘어갔다면, 원래의 배열은 그 값을 유지해야 할 것이다. 결과는 그렇지 않다는 것을 보여 준다.

[포인터로 넘기고, 배열로 쓰기]

포인터(Pointer)는 주소(Address)를 가지는 타입이다. 포인터와 배열은 유사한 점이 많으며, 특히 배열로 선언된 자료구조를 포인터를 사용할 수 있다. 하지만, 포인터를 사용하게 될 경우 실수를 할 경우가 많으며, 코드를 이해하는 것도 배열에 비해 어렵다. 예를 들어, 다음의 코드를 보도록 하자.

```

unsigned int array[ 100 ];
unsigned int* pointer;
...

pointer = array;
array[ 99 ] = 100;      /* 배열을 접근 */
*(pointer + 99) = 100; /* 포인터를 이용해서 배열을 접근 */

```

위의 코드에서 어떤 코드가 더 읽기 쉬운가? 아무 아파 당연히 "array[]"와 같이 배열을 사용하는 것이 좀 더 직관적으로 보일 것이다. 직관적이라는 것은 더 읽기 쉽다는 것이다. 그리고, 결국 읽기 쉬운 것이 "더 좋은 코드"라는 결론이다. 코드를 읽기 쉽게 만드는 것은 간단한 일처럼 보이지만, 실제로는 힘든 일이라는 것을 경험있는 개발자들은 잘 알고 있을 것이다.

함수를 만드는데도 포인터를 사용하는 것이 도움을 준다. 배열의 경우 함수를 선언할 때 다차원 배열이 있다면, 최소한 2차원 이상의 요소에 대해서는 크기를 명시적으로 주어야 선언할 수 있다. 하지만, 포인터를 이용할 경우에는 그것이 필요하지 않다. 즉, 배열을 이용한 함수의 선언은 특정 크기 만을 입력으로 받을 수 있는 한계를 지니지만, 포인터를 이용하면 그런 한계를 명시적으로 가질 필요가 없다.

```

#define WIDTH 100
unsigned int function( unsigned int input[ ][ WIDTH ], unsigned int size ) {
    ...
}

```

```

unsigned int function( unsigned int* input, unsigned int size ) {
    int i;
    ...
}

```

```

for( i = 0; i < size; i ++ ) {
    input[ i ] = ...;
    ...
}
return result;
}

```

위의 코드와 같이 포인터와 배열이 호환이 가능하다는 점을 이용해서 좀 더 유연한 코드를 만들었다는 것을 알 수 있다. 특정 크기를 명시적으로 지정하지 않더라도 쉽게 확장해서 사용할 수 있다는 것이다. 함수를 좀 더 유연하게 만든다는 것은 당연히 함수의 재활용 가능성을 넓혔다고 생각할 수 있으며, 이런 것들을 통해서 "읽기도 쉽고 재활용 하기도 좋은 코드"를 만들어 가는 것이다.

[배열의 크기를 넘길 필요가 없도록 만들기]

앞에서 본 코드들은 배열을 함수의 인자로 전달하기 위해서 배열의 길이를 같이 넘겨주어야 한다는 약점이 있다. 배열은 크기를 알지 못하면 오버플로우(Overflow)를 일으키기 쉽우며, 함수에 배열을 인자로 넘길 경우 항상 그 크기를 같이 알려주어야 한다. 하지만, 배열 자체에 끝을 나타내는 표시를 한다면, 호출되는 함수에서 그 끝을 찾을 수 있어서 크기를 넘겨줄 필요가 없어진다.

```

#include <stdio.h>
#include <stdlib.h>

#define DELIMITER '\0'

void function(char array[]) {
    unsigned int i = 0;

    printf("Array has : ");
    while (array[i] != DELIMITER) {
        printf("%c", array[i]);
        i++;
    }
    printf(".\n");
}

int main(void) {
    puts("Array Delimiter Example 01");

    char array[] = { "HELLO, WORLD!!!" };
    function(array);

    return EXIT_SUCCESS;
}

```

위의 코드는 “function()”함수에서 입력받은 문자를 가지는 배열의 내용을 하나씩 출력한다. 입력으로 주어진 “array[]”에는 명시적으로 구분자(Delimiter)를 주지 않았지만, 문자열의 끝에는 항상 “\0” 들어가기 때문에 프로그램의 실행에는 영향을 주지 않는다.

함수의 호출에서 보듯이, 배열의 원소에 구분자를 사용하는 경우에는 배열의 크기를 명시적으로 함수의 호출시 넘겨줄 필요가 없다. 한가지 주의해야 할 점은 배열의 원소가 절대 구분자를 값으로 가지지 않는다고 보장해야 한다는 점이다. 또한, 구분자를 다루는 것이 누락될 경우에는 프로그램이 엉뚱한 주소 공

간을 접근할 가능성이 있기에 조심해서 사용해야 한다. 이런 것들이 부담이라면 배열의 크기를 명시하는 이전에 사용한 방법을 고려해야 할 것이다.

[배열은 간단하게 사용]

배열(Array)은 자료구조를 저장하기 위해서 사용하지만, 복잡해지면 오류가 발생할 가능성이 높다. 따라서, 이런 배열에 대한 접근은 쉽게 할 수 있는 방법으로만 자료구조를 정의하는 것이 좋다. 될 수 있으면 다차원 배열보다는 1차원 정도 수준에서 배열을 사용하는 것이 좋고, 배열에 접근하는 함수들을 따로 떼어내서 배열과 같이 묶어 사용할 수 있도록 해 주는 것이 좋다. 즉, 배열 자체를 다른 코드와는 분리시키는 것이다. 배열은 언제든 다른 자료구조로 변경될 가능성이 있기 때문이다. 항상 자료구조에 대한 정보는 그것을 직접적으로 접근하는 코드를 제외한, 다른 코드로부터 감추는 것이 좋다. 배열에서 가장 실수를 흔하게 하는 부분은 배열의 인덱스를 잘못 사용하는 것이다. 하지만, 이것도 직접적으로 자료구조에 접근하지 못하게 만들고, 배열의 접근 함수를 따로 정의한다면 쉽게 해결 가능하다.

```
int DataArray[MAX_SIZE] = { 0 };

int getData(int index) {
    if ((index < 0) || (index >= MAX_SIZE)) {
        return 0; /* 이 경우에는 오류로 가정한다. */
    }
    return DataArray[index];
}

int setData(int index, int value) {
    int retva = 0;

    if ((index < 0) || (index >= MAX_SIZE)) {
        return 0; /* 이 경우에는 오류로 가정한다. */
    }
    retval = DataArray[index];
    DataArray[Index] = value;
    return retval;
}
```

위의 코드는 간단히 자료구조로 배열을 선언하고, 배열을 직접적으로 접근하는 API를 같이 정의한 경우이다. 물론, 이렇게 사용하는 것이 성능을 낮춘다고 이야기 할 수 있겠지만, 나중에 API를 "#define"을 사용한 매크로로 변경하거나 Inline 함수로 변경할 수도 있으며, 경계를 검사하는 부분을 디버깅(Debugging)목적으로만 조건부 컴파일을 할 수 있도록 만들어 줄수 있기에 성능상 크게 문제는 되지 않을 것이다.

배열을 이차원으로 사용할 경우에도 마찬가지 방법으로 사용할 수 있지만, 생각을 좀더 구체화 시키기 위해서 좌표값과 같이 주기 보다는 명칭을 사용하는 것이 좋다. 예를 들어, 데이터를 입력한다면 대상이 되는 열과 행에 대해서 이름을 주는 방식으로 함수를 만든다.

```
int ItemData[ MAX_COLUMN, MAX_ROW ] = { 0 };

int setCount( int itemID, int fieldID, int count ) {
    ItemData[ itemID, fieldID ] = count;
}
```

위의 코드에서는 배열의 유효범위를 검사하는 것은 생략했다. 하지만, 위와 같이 2차원 배열을 사용할 때, 각각의 행과 열에 이름을 주고 값을 주는 함수를 만든다면, 사용하는 측에서는 좀더 명확하게 사용을 할 수 있게 된다. 내부적으로는 배열이 아닌 다른 자료 구조를 구현 하더라도 개념적인 일관성은 유지할 수 있을 가능성이 높다.

3차원 이상의 배열을 사용하는 것도 필요한 경우가 있겠지만, 될 수 있으면 자제하는 것이 좋다. 이유는 단순하다. 실수할 수 있는 가능성이 높아지기 때문이다. 물론, 단위 테스트를 통해서 안정적으로 만들 수도 있겠지만, 기본적으로 복잡한 자료구조는 사용하지 않는 것이 코드를 더 쉽게 이해할 수 있도록 만든다. 고차원 배열을 사용할 수록 코드도 더 복잡해질 가능성이 높기에 될 수 있으면 간단한 자료구조를 여러개를 사용하는 방법으로 복잡한 구조를 피할 수 있도록 만드는 것이 좋다. 복잡한 구조를 피하는 것이 코드의 수정도 더 용이하게 만든다.

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE 100
int DataArray[MAX_SIZE] = { 0 };

int getData(int index) {
    if ((index < 0) || (index > MAX_SIZE)) {
        return 0;
    }

    return DataArray[index];
}

int setData(int index, int value) {
    int retvalue = 0;

    if ((index < 0) || (index > MAX_SIZE)) {
        return 0;
    }
    retvalue = DataArray[index];
    DataArray[index] = value;

    return retvalue;
}

int main(void) {
    int storedData = 0;
    setData(10, 100);
    storedData = getData(10);
    printf("The Stored Data is : %d.\n", storedData);
    return EXIT_SUCCESS;
}
```

위의 코드는 간단히 앞에서 만든 배열에 접근하는 API들에 대한 사용법을 보인 것이다. 기존에 있던 데 이터를 알고 싶어하는 경우를 대비해서 `setData()`를 정의하고 있지만, 직접적으로 사용하지는 않았다. 배열의 범위를 점검하는 코드도 삽입되어 있기에, 실수할 가능성은 줄어들었음을 확인할 수 있을 것이다. 배열에서 나타나는 대표적인 오류는, 배열의 범위를 넘어서거나 첫번째 원소의 앞을 접근하는 경우

다. 따라서, 이런 오류를 사전에 점검하지 않는다면 오류를 발생시킬 가능성인 높다. 다음에서 좀더 자세히 알아보자.

[배열의 인덱스(Index)는 자주 문제를 일으킨다.]

C언어에서 배열은 문제를 자주 일으키는 편에 속한다. 특히, 배열에 대한 접근을 "for()"루프와 같은 곳에서 할 때, 배열의 경계를 넘어서는 일이 종종 발생한다. 이런 일이 지속적으로 발생한다는 말은 배열의 사용에 있어서 중요한 점을 생각하지 못하고 있다는 뜻이다. 따라서, 이럴 경우에는 문제가 생기지 않는 방법을 근본적으로 적용해야 한다.

```
#include <stdbool.h>

#define MAX_LENGTH ARRAY_SIZE - 1

static inline bool isEnd(int i) {
    return i <= MAX_LENGTH;
}

...

for( int i = 0; isEnd( i ); i ++ ) {
    do_something();
}
...
```

위의 코드는 배열의 마지막인지는 확인하는 방법을 인라인 함수로 만들었고, 배열의 크기보다 하나 더 작은 크기를 가지고, 항상 배열의 마지막인지를 점검하도록 해 주었다. 물론, 이런 코드가 귀찮아 보일수는 있다. 좀 더 단순화된 방법으로 구현할 수도 있을 것이다. 어쨌든, 이런 방법으로 얻는 효과는 배열에 대해서 접근 오류를 발생시키지 않는다는 것이다.

배열에 대한 연산을 배열을 정의한 곳으로 다 옮길 수 있다면, 자료구조의 노출을 최소화 할수도 있을 것이다. 즉, 배열을 노출하지 않고 연산을 적용하는 것이다. 예를 들어, 배열의 전체 값에 1씩 증가를 시키기를 원한다면, 아래의 코드와 같이 할 수 있을 것이다.

```
#define ARRAY_SIZE 100

typedef struct myType MYTYPE;

struct myType {
    int array[ 100 ];
    void (*inc)( MYTYPE *type );
}
...

#define MAX_LENGTH ARRAY_SIZE - 1

static inline bool isEnd(int i) {
    return i <= MAX_LENGTH;
}

void increaseAll( MYTYPE *type ) {
    int *array = type->array;
```

```

for ( int i = 0; isEnd( i ); i++ ) {
    (*(array + i ))++;
}
}

MYTYPE mytype = {{0}, increaseAll };

...
mytype.inc( &mytype );
...

```

위의 코드는 증가시키는 함수를 배열과 같이 선언해서, 구조체의 내부 함수를 호출하는 방식으로 해결하는 것이다. 이때는 내부에 배열을 사용하는지 알 필요없이, 단순히 정의된 함수를 이용해서 원하는 일을 처리하도록 맡기는 것이다. 위의 코드에서는 "MYTYPE"이라는 것을 이용해서, 외부에서 사용할 변수의 타입을 알려주고, 내부적으로는 해당 타입에 대한 함수의 포인터를 이용하는 방식으로 구현했다. 물론, 함수의 포인터를 다시 외부에 노출하지 않기 위해서, 외부에 노출되는 함수(Wrapper Function)를 따로 구현할 수도 있을 것이다.

이와 같이해서 얻는 효과는 내부 자료구조에 대한 노출을 막아서, 향후 변경에 대한 외부 코드에 주는 영향을 줄일 수 있다는 점이다. 즉, 내부 자료구조가 어떻게 구현 되었는지를 모르고도 충분히 사용할 수 있다. 물론, 얻는 이점에 대해서 희생해야하는 점도 있다. 즉, 함수의 호출을 통해서 접근한다는 점이다. 하지만, 이런 인터페이스를 이용한 자료접근은 얻는 점이 더 많기에 충분히 고려해 볼 만 하다.

[배열 사용시 오버플로우(Overflow)의 주의]

배열(Array)는 동일한 자료타입을 가지는 원소들을 묶어놓기 위해서 사용한다. 문제는 배열의 선언된 값과 실제 배열의 길이가 다르다는 점이다. 즉, 배열은 "0번째: 부터 시작해서, "선언된 값 - 1"까지만 사용될 수 있다. 그 이상이나 그 이하로 접근하는 것은 다른 데이터를 손상시키거나, 프로그램을 예측할 수 없는 상황에서 오류를 발생시킬 수 있다. 따라서, 배열에 대한 직접적인 접근보다는 좀 더 안정적인 접근을 충분히 고려해 볼 수 있을 것이다.

```

#define MAX_LENGTH 100
int array[ MAX_LENGTH ] = { 0 };

static inline int getElement( unsigned int index ) {
#ifndef __DEBUG__
    if ( ( index < 0 ) || ( index >= MAX_LENGTH ) ) {
        ASSERT("Array access violation occurred!!!");
    }
#endif
    return array[ index ];
}

static inline void setElement( unsigned int index, int value ) {
#ifndef __DEBUG__
    if ( ( index < 0 ) || ( index >= MAX_LENGTH ) ) {
        ASSERT("Array access violation occurred!!!");
    }
#endif
    array[ index ] = value;
    return;
}

```

```
}
```

위의 코드는 간단히 배열에 대한 접근을 담당하는 함수를 만든 것이다. 물론, 모든 배열에 대해서 이런 것들을 만드는 것은 어려울 수 있다. 하지만, 내부 자료구조와 코드를 분리하는 방법으로서 제공할 수 있는 인터페이스를 만든다면, 이야기는 달라질 수 있다. 즉, 배열을 사용하는 입장에서 내부 자료구조가 배열인지 아닌지를 알지 못하게 만드는 방법이다.(여기서는 “index”를 제공해서 어디서 원하는 값을 가져올지, 혹은 어디에 값을 설정할지를 알수있도록 했지만, 내부에서 자료구조의 저장 방법이 해쉬를 사용하는 경우도 가정해 볼 수 있다.)

위의 코드에서는 배열이라는 것을 짐작할 수 있는 부분이 있지만("Element", "Index"), 그런 부분들이 없는 인터페이스를 제공하는 것이다. 예를 들어, "getValue()" 나 "setValue()"와 같은 함수를 제공한다면, 사용하는 측에서는 단순히 값을 읽어내고, 값을 설정하는 용도로만 사용할 수 있는 자료구조가 있다고 판단할 것이다. 물론, 내부적으로 자료구조를 구현하는 것은 담당자의 몫이다.

C언어에서는 배열이나 포인터를 잘못 이용해서 발생하는 오류가 많기에, 이런 부분들에 대한 적극적인 방어코드가 필요하다. 디버그 시에만 사용할 수 있는 방어 코드들을 이용해서("#ifdef __DEBUG__ ... #endif"), 제대로된 검증을 거치도록 만드는 것이 중요하다. 이런 부분들은 단위 테스트로 케이스를 만들어서 보완해도 되지만, 그렇지 못한 경우도 대비하는 것이 좋다.

[구현은 숨기고 선언은 보여주기]

자료구조는 직접적으로 접근하는 것을 최대한 막아야 한다. 그리고, 자료구조에 대한 접근을 반드시 해야할 경우라면, 제공해주는 인터페이스(Interface)를 사용하도록 만들어야 한다. 이유는 코드의 변경이 발생할 경우, 다른 부분과의 의존성을 낮출 수 있기 때문이다. 즉, 구현과 인터페이스를 분리시키는 원리를 이용해서 구현하는 것이 좋다. 특히, 사용자가 정의한 자료구조들은 접근하는 함수와 자료구조 자체를 같이 정의해서 하나의 모듈로 만들어주면, 내부 구현에 대한 정보를 최대한 외부로 보여주지 않게되어, 독립적으로 수정될 수 있도록 해준다(의존성이 줄어든다).

```
#ifndef CTYPE_H_
#define CTYPE_H_

typedef struct AbstractType myType;

myType *Allocate(void);

#endif /* CTYPE_H_ */
```

위의 코드는 자료구조의 사용자가 필요한 정보만을 보여주기 위해서 정의한 헤더 파일이다. 즉, 자료구조를 사용하기를 원하는 사용자들은 이 헤더 파일만 보고 코딩을 해야한다. 내부적인 구조를 파악하고, 그것을 직접적으로 접근하는 것은 허락되지 않는다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "CType.h"

struct AbstractType {
    int ArrayData[100];
};
```

```

myType* Allocate(void) {
    myType *retval;
    retval = (myType*) malloc(sizeof(struct AbstractType));
    if (retval == NULL) {
        return NULL;
    }
    memset((void* ) retval, 0, sizeof(struct AbstractType));
    return retval;
}

```

위의 코드는 해당 자료구조를 사용하기 위해서 필요한 API를 정의하는 구현 파일이다. 구현 파일에 자료구조의 내부 구현도 포함되며, 이를 접근해서 사용하는 함수들도 같이 정의한다. 자료구조의 사용자는 자료구조의 내부를 알 필요없이 단순히 제공되는 API만을 가지고 접근할 뿐이다.

```

#include <stdio.h>
#include <stdlib.h>
#include "CType.h"

int main(void) {
    myType* memory;

    puts("Abstract Data Type");
    memory = Allocate();
    free(memory);

    return EXIT_SUCCESS;
}

```

위의 코드는 자료구조를 접근하는 API를 이용해서 어떻게 해당 자료구조를 사용하는지를 보여준다. 여기서, 한 가지 개선해야 할 점이 있다면, 자료구조의 내부에서 메모리의 할당이 있었지만 해제는 없다는 점이다. 만약 메모리의 할당이 있었다면, 해제를 위한 인터페이스도 같이 가지고 있는 것이 좋다. 자원의 할당이 있었던 곳에서 가장 가까이에서 할당된 자원을 해제를 하는 것이 좋기 때문이다. 그렇게하지 않는다면, 자원 누수(Resource Leak)라는 문제가 발생할 가능성이 있다.

```

#ifndef CTYPE_H_
#define CTYPE_H_

typedef struct AbstractType myType;

myType *Allocate(void);
void Free(myType* resource);

#endif /* CTYPE_H_ */

```

위 코드는 자료구조의 해제를 위해서 추가한 API를 넣어본 것이다. 이제는 이 두개의 API가 쌍으로 사용되기에 자료구조의 접근자는 반드시 쌍으로 사용해야 한다는 것을 알아야 한다. 더 좋은 방법이 있다면, 자료구조의 사용자가 알아야 할 API간에 상관관계가 없는 것이 더 좋다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "CType.h"

struct AbstractType {
    int ArrayData[100];
};

myType* Allocate(void) {
    myType *retval;
    retval = (myType*) malloc(sizeof(struct AbstractType));
    if (retval == NULL) {
        return NULL;
    }
    memset((void* ) retval, 0, sizeof(struct AbstractType));
    return retval;
}

void Free(myType *resource) {
    free(resource);
}
```

위의 코드에서는 단순히 할당받은 자료구조를 해제하는 것으로 구현했다. 나중에 자원의 할당이 달라진다면, 그것에 맞춰서 해제도 구현 하면 될 것이다.

```
#include <stdio.h>
#include <stdlib.h>
#include "CType.h"

int main(void)
{
    myType* memory;

    puts("Abstract Data Type");
    memory = Allocate();
    Free( memory );

    return EXIT_SUCCESS;
}
```

이제는 자료구조의 사용하는 측에서는 해제를 담당하지 않는다. 따라서, 자료구조를 직접적으로 접근하는 함수들의 몫이되고, 사용자는 그냥 해당 API를 호출하는 것으로 원하는 목적을 수행할 수 있게된다. 코딩을 진행하면서 느꼈겠지만, 내부적인 구조에 의존적인 코드를 사용자 측에서 구현하지 않게되면, 추가적인 내부 자료구조의 변경이 발생하더라도 사용자 코드(Client Code)의 변경은 발생하지 않는다. 이렇게 구현하는 것이 미래에 있을 변경에 대비해서 조금 더 유연한 구조를 가지고 만드는 것이다.

[인터페이스를 이용한 구현의 분리]

구현과 인터페이스를 분리하는 이유는 특정 구현에 의존적이지 않은 코드를 만들기 위한 것이다. 즉, 내부적인 구현에 의존 할수록 변화의 영향을 받을 가능성이 높으며, 코드의 수정이 의도하지 않은 버그를

만들어낼 수도 있다. 따라서, 특정 구현에 의존적인 방법으로 코딩하는 것은 권장하는 것이 아니며, 될 수 있으면 의존성을 가지지 않은 코드를 만들어야 유지보수와 확장이 쉬워진다.

일반적으로 사용하는 구현과 인터페이스를 분리하는 방법은 API를 헤더 파일에 정의하고, 구현을 C파일에 하는 것이다. 보여주는 것은 헤더 파일만 보여주게 되고, 구체적인 구현에 대해서는 컴파일된 ".o" 와 같은 파일로 제공하거나, 혹은 라이브러리로 만들어서 제공하는 것이다. 물론, 이것도 훌륭한 방법이다. 실제 구현에서 활용하기 위해서는 특정 자료구조에 의한 의존성을 낮추고, 파일간의 연결 고리를 단방향으로 만드는 것이 좋다.

즉, 하나의 파일이 다른 파일을 접근할 때, 접근되는 파일에서 접근하는 파일로 역으로 접근하는 것은 좋지 않다. 즉, 두 개의 파일간에 순환적인 고리가 만들어질 가능성이 높다. 이런 경우는 두 개의 파일이 맡은 각각의 역할이 중복되어 있거나 분명히 나누어지지 않은 경우에 생기게 되며, 이때는 역방향으로 접근하는 함수나 자료구조에 대해서 재 설계를 통해서 나누어주거나, 다른 파일로 옮겨서 한 방향으로 접근하도록 만드는 것이 해결 방법이다.

```
/* File X */
#include <stdio.h>

void function_A(void) {
    printf("I am function A\n");
    function_C();
}

void function_B(void) {
    printf("I am function B\n");
}

/* File Y */
#include <stdio.h>

void function_C(void) {
    printf("I am function C\n");
    function_B();
}
```

만약, 위와 같이 "function_A()"가 다른 파일에 있는 "function_C()"를 호출하고, "function_C()"가 호출한 측의 "function_B()"를 호출한다면, 이는 두개의 파일이 서로 의존성이 강하게 얹여있는 순환 호출 관계를 만드는 것으로, 둘 중 어떤 파일에 변경을 받았을 때 다른 파일도 변경에 대해서 자유롭지 못하게 만들 가능성이 있다. 이때는 "function_B()"를 "File Y"로 옮겨주는 것이 해결책이다. 이와 유사경우처럼, 파일들 간의 의존성도 중요하게 다루어야 하며, 디렉토리 간의 순환 참조관계도 역할의 혼돈에서 발생할 수 있다. 이때는 해당하는 파일이나 함수등이 정말 그 디렉토리에 존재해야 하는지 책임과 역할을 명확하게 구분해 줄 수 있어야 한다.

구현과 인터페이스를 분리하기 위해서, 인터페이스 역할을 하는 계층을 디렉토리로 분리하고, 그 하위에 구체적인 개별 구현에 맞는 디렉토리를 만드는 것도 유지보수나 이식성을 높이도록 만드는 방법으로 사용할 수 있다. 즉, 특정 CPU에 의존적인 것들은 특정 하위 디렉토리로 만들고, 그 디렉토리 위에 상위의 모듈에서 접근하는 인터페이스를 만드는 것이다. 인터페이스에서는 특정 CPU에 의존적인 기능들만을 제공하는 API를 선언하고, 각각의 API는 특정 CPU에서 제공하는 기능을 이용하도록 다시 구현하는 것이다.

이처럼 디렉토리의 계층을 명확히 해주면 나중에 어디를 고쳐야 할지가 명확해지며, 구체적인 구현의 변경이 발생하더라도 그것을 이용하는 코드들은 변경하지 않아도 된다. 따라서, 다양한 CPU나 플랫폼(Platform)등으로 이식하는 경우에는 이런 것들이 필요하다. 물론, 몇 단계를 거치는 함수의 호출 길이가 길어질 가능성은 있으나, 그것으로 인한 오버헤드는 무시할 수 있을 만큼 작을 것이며 오히려 얻는 효과가 더 클 것이다.

자료구조에 의존적이지 않게 만들기 위해서는 자료구조에서 얻기 원하는 데이터를 API로 추출하는 방식으로 변경해야 한다. 즉, 즉각적인 자료구조에 대한 접근 방법보다는 특정 API를 통한 접근방식으로 대체하는 것이다. 매크로(Macro)나 인라인 함수(Inline Function)등을 이용해서 이를 구현한다면, 즉각적인 접근방법에 비해서 그렇게 심한 오버헤드도 없을 것이며, 적절한 컴파일러의 최적화 기능을 이용한다면 효과적인 분리와 성능간의 편차(오버헤드)도 줄일 수 있을 것이다.

```
struct Complex {
    char *name;
    int value;
    ...
}

char *getNameofComplex(struct *Complex) {
    return Complex->name;
}

int *getValueofComplex(struct *Complex) {
    return Complex->value;
}
```

위의 구현에서 중요한 점은 API를 사용하는 측에서 본다면, 내부의 자료구조에의 구현에 대해서는 모른다는 점이다. 즉, 단순히 알고 있는 자료형의 이름만을 이용해서 접근하기 때문에, 나중에 구체적인 자료구조의 변경이 발생하더라도 코드를 수정할 가능성은 줄어들게 된다는 뜻이다. 이것은 간단한 원리지만, 복잡한 프로그램을 구현하는데 있어서는 중용한 개선점이 될 수 있다. 될 수 있으면 내부적인 구현 정보를 외부에 보여주지 않는 것이 좋은 코딩의 기초다. 구체적인 것에 대한 의존성을 줄이는 것이 결국 코드를 유지보수하고 확장하는 비용의 경감을 가져오기 때문이다.

[헤더 파일의 사용]

헤더 파일은 프로그램에서 사용할 자료구조 및 형(Type)에 대한 정의를 담고 있는 파일이다. 일반적인 ".c"파일과 마찬가지로 헤더 파일도 다른 헤더 파일을 참조할 수 있으며, 이런 참조 관계가 복잡하면 코드를 어렵게 만드는 요인이 될 수 있다. 따라서, 헤더 파일간의 의존성(Dependency)를 낮게 가져가는 것도 중요한 코딩 활동이라고 볼 수 있다.

헤더 파일들 간의 의존성을 약하게 하기 위해서는 헤더 파일을 만드는 규칙을 정할 필요가 있다. 대부분의 헤더 파일은 여러번 프로그램에 포함될 가능성이 있기 때문에 아래와 같은 형식을 주로 많이 사용한다.

```
#ifndef REMOVESWITCH_H_
#define REMOVESWITCH_H_

enum MODEL_TYPE {
    A_MODEL = 0,
    B_MODEL,
    C_MODEL,
```

```

};

/* 이 부분도 감추는 것이 좋다.*/
typedef struct _MODEL_TYPE {
    int code;
    char *name;
    int (*handler)( struct _MODEL_TYPE *model );
} MODEL_TYPE;

extern MODEL_TYPE *createModel( int model_type );
extern int model_handler( MODEL_TYPE *model );

#endif /* REMOVESWITCH_H_ */

```

즉, 파일의 앞 부분과 뒷 부분에 "#ifndef ... #define ~~~ #endif"를 사용해서 두 번 이상 포함되더라도 문제가 생기지 않도록 만들어준다. 사용할 자료구조에 대한 정의와 다른 모듈에 보여줄 외부에서 정의된 (*extern*) 함수의 원형(Prototype)을 알려준다. 이 함수들은 주로 정의된 자료구조를 사용하기 위해서 필요한 인터페이스를 제공하며, 데이터 타입은 참조를 위해서 보여주게 된다. 사실, 사용하는 측에서 알아야 할 정보는 데이터 타입의 내부적인 구조가 아니라, 단순히 인터페이스에 한정시키는 것이 자료구조에 대한 의존성을 줄이는데 도움이 된다.

추가적으로 헤더 파일이 다른 헤더 파일을 “Include”하게 되면, 컴파일러가 컴파일하기 위해서 참조해야 할 파일들이 늘어나게 되며, 이로 인해서 컴파일 시간 자체가 길어지게 된다. 또한, 조그만 코드의 수정에 의해서도 전체 파일들이 다시 컴파일되어야 하는 오버헤드도 있기 때문에, 될 수 있으면 헤더 파일내에 다른 헤더 파일을 “Include”시키지 말아야 한다.

[C구현 파일 나누기]

여러 사람이 개발하는 경우에도 구현 파일을 여러 개로 나누는 것이 필요하지만, 파일을 나누는 것은 생각을 나누는 것과 동일하게 보아야 한다. 소프트웨어가 복잡하다고 이야기하는 것은, 그 복잡함을 어떻게 해결 하느냐에 따라 좋은 코드와 그렇지 않은 코드로 나뉘어진다는 것을 알 수 있다. 구현 파일을 나누는 것은 복잡한 문제를 작은 문제로 쪼개는 방법으로 해결책을 제공한다.

이렇게 쪼개야 하는 이유는 구해야 할 전체적인 해결책에 대한 추상적이고 작은 해결책들을 제공하기 위함이다. 하나의 구현 파일은 한가지의 역할에 충실해야 하는 것이 원칙이다. 만약, 일관되게 한가지 일을 수행하지 못한다면 구현 파일을 더 작게 나누어야 한다. 함수가 하나의 “기능”만을 수행하듯, 하나의 구현 파일에 묶여있는 함수들은 일관되게 하나의 목적을 수행하는 것들로 이루어져야 한다.

```

#include <stdio.h>
#include <stdlib.h>

unsigned int factorial(const int nth) {
    if (nth <= 0) {
        return 0;
    }

    unsigned int result = 1;

    for (int i = 1; i <= nth; i++) {
        result = result * i;
    }
}

```

```

    return result;
}

int main(void) {
    puts("Factorial Demo"); /* prints Factorial Demo */

    printf("The value of %dth factorial is : %ld\n", -1, factorial(-1));
    printf("The value of %dth factorial is : %ld\n", 0, factorial(0));
    printf("The value of %dth factorial is : %ld\n", 1, factorial(1));
    printf("The value of %dth factorial is : %ld\n", 2, factorial(2));
    printf("The value of %dth factorial is : %ld\n", 3, factorial(3));
    printf("The value of %dth factorial is : %ld\n", 5, factorial(5));
    printf("The value of %dth factorial is : %ld\n", 10, factorial(10));
    return EXIT_SUCCESS;
}

```

위의 코드는 팩토리얼(Factorial) 값을 구한다. 코드의 테스트를 위해서 경계 조건에 해당하는 값을 이용해서 제대로 코드가 동작하는지도 검증했다. 이제 이 코드를 분리해서 몇 개의 파일로 나누도록 한다. 원칙은 하나의 구현 파일에 대해서 하나의 헤더 파일을 만든다.

```

#ifndef __LIBFACTORIAL_H__
#define __LIBFACTORIAL_H__

#ifdef __cplusplus
extern "C" {
#endif

unsigned int factorial( const int nth );

#ifdef __cplusplus
}
#endif /* __cplusplus */
#endif /* __LIBFACTORIAL_H__ */

```

위의 코드는 "LibFactorial.h" 파일을 보여준다. "include"시켜주기 위해서 선언된 것들을 반복할 되지 않도록 했으며("#ifndef __LIBFACTORIAL_H__"), C++에서도 사용할 수 있도록 해 주었다("#ifdef __cplusplus"). 함수는 구현을 제외하고 선언만 정의했다.

```

#include "libFactorial.h"

unsigned int factorial( const int nth ) {
    if ( nth <= 0 ) {
        return 0;
    }

    unsigned int result = 1;
    for( int i = 1; i <= nth; i++ ) {
        result = result * i;
    }

    return result;
}

```

```
}
```

위의 코드는 "LibFactorial.c" 파일을 보여준다. "LibFactorial.h"에서 선언한 함수를 구현했으며, 자신이 정의하고 있는 함수를 선언한 "LibFactorial.h" 파일을 "include"해 주었다. "-std=c99"를 컴파일 옵션으로 추가해 주어야 컴파일 오류가 발생하지 않을 것이다("unsigned int result = 1;"에서).

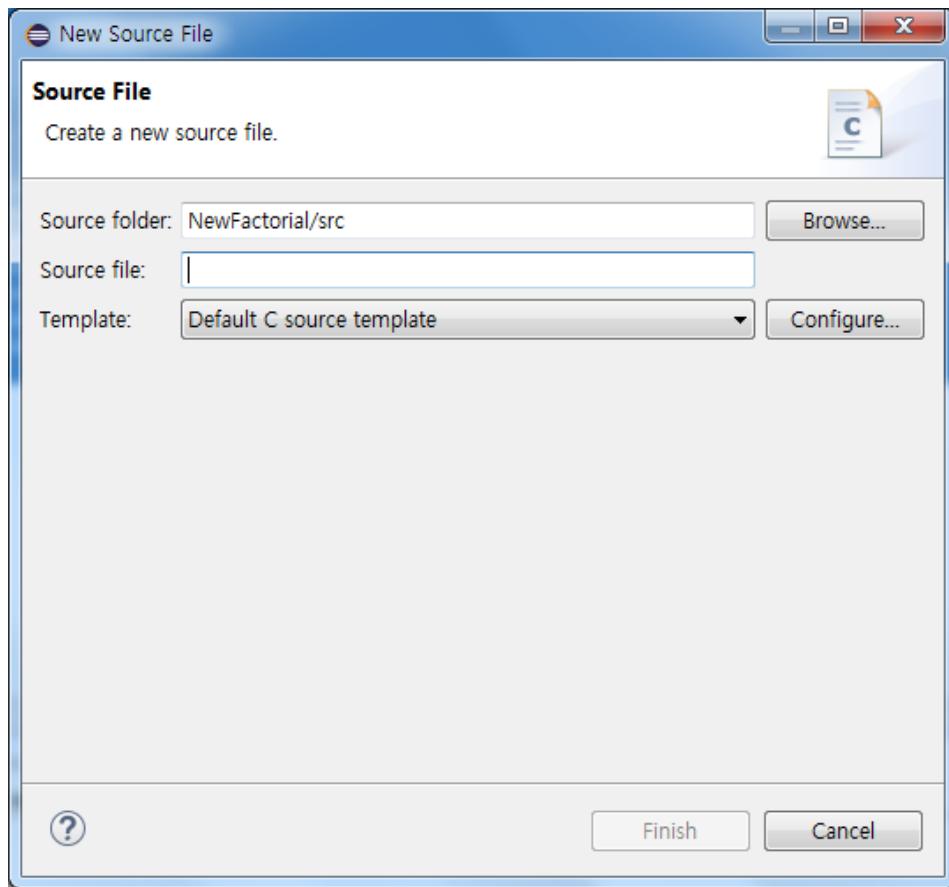
```
#include <stdio.h>
#include <stdlib.h>

#include "LibFactorial.h"

int main(void) {
    puts("Factorial Demo");

    printf("The value of %dth factorial is : %ld\n", -1, factorial( -1 ));
    printf("The value of %dth factorial is : %ld\n", 0, factorial( 0 ));
    printf("The value of %dth factorial is : %ld\n", 1, factorial( 1 ));
    printf("The value of %dth factorial is : %ld\n", 2, factorial( 2 ));
    printf("The value of %dth factorial is : %ld\n", 3, factorial( 3 ));
    printf("The value of %dth factorial is : %ld\n", 5, factorial( 5 ));
    printf("The value of %dth factorial is : %ld\n", 10, factorial( 10 ));
    return EXIT_SUCCESS;
}
```

위의 코드는 분리된 파일로 구현한 팩토리얼 함수를 호출할 수 있도록 "LibFactorial.h" 헤더 파일을 추가한 것을 볼 수 있다. Eclipse에서는 프로젝트에 파일을 추가하기 위해서 "File->New->Source File"을 선택하도록 한다.



"Source file :" 부분에 필요한 파일의 이름을 넣어주면 된다. 기본적으로 이렇게 추가된 프로젝트 파일의 경우는 빌드(Build)시에 같이 컴파일되며, 실행 가능한 코드를 생성하기 위해서 필요한 함수나 변수의 주소를 찾기 위한 링크(Link) 과정에서 사용된다. 따라서, 앞에서 파일을 나누어주는 것은 실행 파일을 생성하는데 별다른 영향을 주지 않음을 알 수 있다. 대신에 파일들을 자신이 하는 역할에 따라 나누어 주고, 한 가지의 책임에 충실히 코드만을 담도록 해서, 이해하기가 조금 더 쉬운 코드로 바뀌었음을 알 수 있다.

짧은 프로그램을 나누어서 할 필요가 없다고 생각할 지 모르지만, 이것은 습관에 관련된 것이다. 습관적으로 이러한 코딩 연습을 하는 것이 실무에서는 많은 도움이 될 것이다. 그리고, 더 중요한 점은 파일들로 나누는 기준에 대한 연습을 꾸준히 해야 한다. 결국 코딩은 소프트웨어로 해결해야 할 복잡한 문제를 더 작은 문제로 나누어서 해결한 후, 이를 합쳐서 더 큰 해결 방안을 찾아가는 과정이기 때문이다.

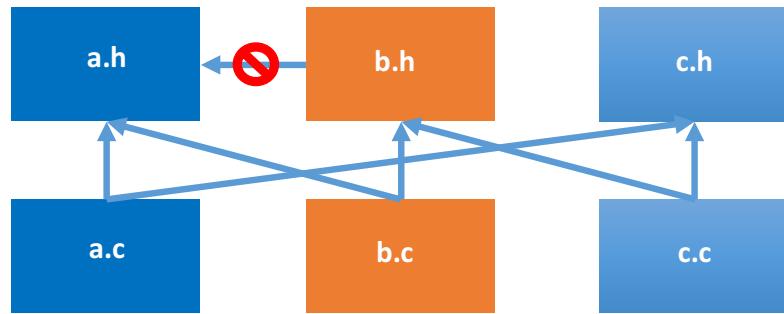
[헤더(Header) 파일의 위치와 내용]

헤더 파일은 다양한 구현 파일(.C)에서 사용된다. 따라서, 어디에 두어야 제대로 된 곳인지 정하기가 어렵다. 먼저, 헤더 파일의 크기를 너무 크게 잡지 않는 것이 좋다. 예를 들어, 하드웨어의 레지스터(Register)들을 접근한다면, 함께 접근될 가능성이 있는 것들만 묶어서 따로 헤더 파일로 정리하는 것이 좋다. 같이 묶어 둔다면 헤더 파일의 크기는 증가 할 것이고, 다양한 코드에서 헤더 파일을 접근해서 구조적인 문제를 유발하기 쉽다. 즉, 계층적으로 분리된 코드를 만들기 위해서는 작은 크기로 함께 묶어야 할 부분들만 헤더 파일로 뽑아내서 관리해야 한다.

header.h --> header_A.h, header_B.h

또는,

header.h --> header_A.h, header_B.h --> header_C.h (Common part of A and B header file : 헤더 A와 헤더 B의 공통 부분을 뽑아서 헤더 C파일을 만들었다.)



헤더 파일의 위치를 정하는 것은 사용자와 제공자의 역할을 정확히 이해해야 한다. 일반적으로 서비스를 제공하는 측에서 접근을 위해서 필요한 정보를 헤더 파일에 명시한다. 따라서, 사용자 측에서 제공자의 헤더 파일의 내용을 가지고 있을 필요는 없다. 헤더 파일의 위치는 그것을 직접적으로 필요로 하는 구현 파일들이 있는 곳이 최적의 위치다. 간접적으로 사용하는 경우에는 "#include"를 사용해 포함하면 되며, 이는 사용자의 계층에서 제공자로 접근하는 형태이기에 계층화 위반을 발생시키지 않는다.

만약, 아래와 같이 헤더 파일내에 다른 헤더 파일을 "#include"해야하는 경우라면, 구현 파일로 위치를 이동시켜주는 것이 좋다. "#include"순서는 바탕이 되는 자료구조가 무엇이 필요한지를 결정할 수 있으면 된다.

```
/* header_C.h */
#include "header_A.h"
#include "header_B.h"
...
```

```
/* header_C.h */
/* No header file included here!!! */
...
```

```
/* implementation_C.c */
#include "header_A.h"
#include "header_B.h"
#include "header_C.h"
...
```

헤더 파일 간에도 의존성이 발생할 수 있다. 즉, 하나의 헤더 파일에서 다른 헤더 파일을 접근하는 경우다. 특히, 계층화 구조를 위반하는 사례가 헤더 파일에서도 많이 발생하는 이유가 그 때문이다. 서로 계층은 다르지만, 하위 계층의 헤더 파일이 상위 계층의 헤더 파일을 "#include"라는 경우라고 볼 수 있다. 이때는 필요한 정보가 상위 계층에 있어야 하는 것이 아니라 오히려 하위 계층에 있어야 한다는 것을 의미하기에, 헤더 파일의 위치를 옮기거나 사용하는 것만 분리해서 따로 헤더 파일을 만들어 하위 계층으로 옮겨주는 것이 옳다.

될 수 있으면 헤더 파일의 내부에서 다른 헤더 파일을 "#include"하지 않도록 해야 한다. 가능한 구현 파일에서 필요한 헤더 파일을 먼저 "#include"해야 한다. 헤더 파일들의 중복 포함 문제를 해결하는 것은 이미 다루었듯이, "#ifndef __XXX_H__ ~ #define __XXX_H__ #endif"와 같이 해결할 수 있다. 또한 헤더 파일 내부에 자료구조를 정의하기 보다는, 구현 파일로 옮기고 해당하는 타입만 보여주는 것

이 좋은 선택이다. 사용하는 측에서는 내부 자료구조의 구현까지 상세하게 알 필요는 없기 때문이다.

```
/* header.h */
typedef struct my_structure MY_STRUCTURE;

MY_STRUCTURE init_function( void );
void do_function( MY_STRUCTURE *pointer_to_my_structure );

/* implementation.c */
#include "header.h"

typedef struct my_structure {
    char *name;
    unsigned int age;
    ...
} MY_STRUCTURE;
```

물론, 타입만 정의하면 인터페이스의 호출시 추가적인 파라미터를 명시해야 하는 오버헤드가 따를 수 있다. 하지만, 정보를 제한해서 얻는 의존성의 감소는 충분한 이유를 제공해 줄 수 있을 것이다. 파라미터의 개수가 늘어나더라도 특정 수 이상만 사용하지 않으면(5개 이상), 성능상의 오버헤드는 줄일 수 있다. 내부 함수는 당연히 헤더 파일에 선언할 필요가 없으며, 구현 파일의 내부로 다 들어가야 한다. 코드를 정리하지 않으면, 내부 함수 까지도 종종 외부로 공개 되는 경우가 있으며, 이는 코드의 독립성을 약화시키는 원인이 될 수 있다.

[조건부 컴파일 줄이기]

코드가 갈아지고 이해하기 힘들어지는 이유는 코드를 처음 만들 때의 목적(혹은, 지원범위)를 넘어서는 지속적인 변경이 발생하기 때문이다. 따라서, 이를 적절하게 관리하지 못하면 급속도로 코드가 복잡해지고 버그를 유발할 가능성은 높아지게 된다. 복잡한 코드는 개발자가 이해하고 수정하는데 필요한 시간을 더 길게 만들어, 야근과 주말 특근의 주요한 원인으로 작용하기도 한다. 따라서, 이런 복잡함을 유도하는 다양한 언어에서 지원하는 도구들은 적절한 상황에서 사용되어야 하며, 무분별하게 사용하면 그것으로 인해 스스로 함정에 빠지게 된다. "#ifdef ~ #endif"도 그 중 하나이며 지나치게 사용하면 코드의 이해도 만 떨어뜨리고, 오류를 유발할 가능성만 키우게 된다.

조건부 컴파일이 늘어나면 늘어날수록 코드는 점점 길어지게 되며, 생각의 흐름은 끊어지기 마련이다. 물론, 이렇게 써야 하는 경우도 있다. 만약, 조건부 컴파일의 범위가 짧고 거의 변경이 발생하지 않는다면, 사용해도 무방할 것이다. 코드를 읽는데 부담을 주지 않을 정도라면 상관없다는 것이다. 하지만, 코드를 단순히 참고하기 위해서 이런 식으로 컴파일 되지 않는 사용되지 않는 코드를 남겨두는 것은 옳지 않다. 가능한 코드는 깔끔한 상태로 유지하려고 노력해야 하며, 코드를 읽는 사람이 실수할 가능성을 줄여주는 것이 좋다.

따라서, 만약 이런 식의 분기(Branch)가 필요하다면, 차라리 파일로 따로 만드는 것을 고려해볼 수 있을 것이다. 컴파일할 때는 해당하는 파일을 명시적으로 빌드(Build)될 수 있도록 만들어주면 된다. 아래와 같이 조건부 컴파일에 들어가야 할 코드들을 묶어서 하나의 파일로 분리한 후, 이를 컴파일러의 입력으로 줄 수 있다.

```
#ifdef __XXX__
obj-y += XXX.c
...
#endif
```

```
#ifdef __YYY__
obj-y += YYY.c
...
#endif

#ifndef __ZZZ__
obj-y += ZZZ.c
...
#endif

all :
${CC} obj-y -o a.out
```

조건부 컴파일은 컴파일 옵션 설정을 어렵게 만든다. 모델의 분기나 특정 기능의 분기를 위해서 컴파일 옵션들은 늘어나기 마련이며, 이는 코드를 잘못 빌드하게 되는 원인이 될 수도 있다. 또한, 옵션들을 기억하고 관리하는 문제도 발생하게 되며, 관련없는 코드들이 한 덩어리로 관리되는 원인이 되기도 한다. 따라서, 조건부 컴파일은 제한된 경우에 한정해서 사용하는 것이 좋으며, 예를 들어 디버그(Debug)나 특정 기능의 테스트 목적 정도만 활용되어야 할 것이다. 충분히 목적을 달성한 이후에는 해당 옵션을 제거하거나 코드를 분리시켜 관리할 수 있도록 해야 할 것이다.

```
void (*interface_function)( void );
...

void default_function( void ) {
    /* Do_nothing!!! */
}

void init_interface( void (*func)() ) {
    ...
    interface_function = func;
}
```

실행 시간(Run-Time)에 모델 분기를 해야할 경우에는 초기화 시에 필요한 인터페이스를 생성하는 방법을 사용할 수 있다. 즉, 특정 모델에서 제공하는 기능이 필요하다면, 해당 부분을 인터페이스로 분리하고 초기화에서 설정하는 방법이다. 만약, 그 기능을 사용하지 않는 제품이라면, 아무런 일도 하지 않는 코드로 대체할 수 있도록 기본(Default) 인터페이스를 제공할 수도 있다.

실시간에 변경을 해야하는 경우는 특정 시스템의 상태마다 다른 행동을 보여야 하는 경우이며, 이는 프로토콜이나 이벤트(Event) 등에 활용할 수 있다. 이 때도 조건문은 될 수 있으면 사용하지 않는 것이 좋으며, 특히 반복적으로 유사한 형태로 등장하는 "switch()"문은 좋은 해결 방법이 아니다. 조건부 컴파일과 마찬가지로 조건문도 될 수 있으면 아껴서 사용해야 한다.

[조건부 컴파일의 제거]

코드를 읽을 때 한번은 경험 했겠지만, 조건부로 컴파일되는 코드들은 이해를 방해하는 것이 많다. 코드가 지나치게 길어지는 경향이 생기거나, 혹은 사용되지 않은 코드를 여전히 남겨두는 경우도 생긴다. 사실 이런 코드들은 제거하는 것이 현명하다. 이미 버전 관리시스템을 사용하고 있는 상황이라면 이런 부분들을 과감하게 정리해 볼 수 있을 것이다. 그리고, 그렇게 특정 상황에 맞게 조건을 만들어서 컴파일되는 코드라면, 인터페이스를 만들어서 분리된 파일로 두는 것이 좋을 것이다.

```
#ifdef MODEL_A           /* 전체 코드에 여러번 등장한다. */
```

```

        do_something_X();
#else
        do_something_Y();
#endif

.....
void (*do_something)( void );
...

void initialize_system( void ) { /* 시스템 초기화 시에 한번만 실행된다. */
#endif MODEL_A
    do_something = do_something_X;
#else
    do_something = do_something_Y;
#endif
}

...
do_something();           /* 조건부 컴파일 없이 그냥 호출해도 된다. */

```

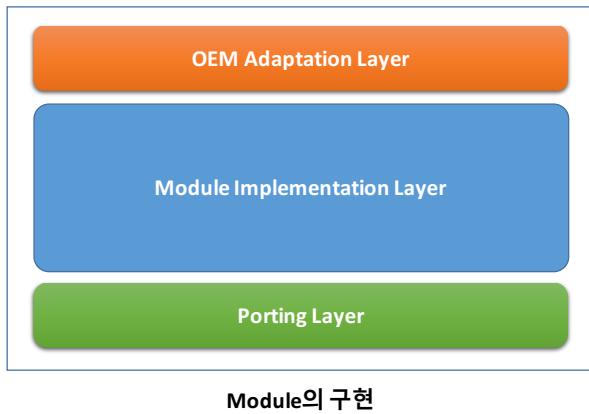
이와 같은 코드들이 늘어나는 것은 제품 모델의 분기나 특정 함수의 테스트, 기존의 코드를 남겨두고 싶은 경우 등등 다양하게 있겠지만, 코드를 점점 더 이해하기 힘들게 만들기는 마찬가지다. 따라서, 만약 이런 부분이 필요하다면, 함수 포인터와 같은 것을 만들거나 매크로 등을 정의해서 함수의 이름을 대체하는 방법으로 제거할 수 있을 것이다.

앞의 예제를 보면 처음에는 같은 것이라고 생각할지도 모른다. 하지만, 여기서 보여주는 것은 코드의 일부일 뿐이다. 만약, "#ifdef ~#else ~#endif"가 반복적으로 나오는 상황이라면, 시스템의 초기화 시에만 설정해주고, 나머지 코드들에서는 조건부 컴파일을 제거할 수 있기 때문이다. 따라서, 다른 코드를 들에 발생할 수 있는 중복의 문제를 제거함과 동시에 좀더 읽기 쉽고 독립적인 코드를 만들어 갈 수 있는 것이다.

"#if 1 ~ #else ~ #endif"와 같이 컴파일되지 않는 코드를 남겨두는 것도 좋은 습관은 아니다. 수정된 코드가 제대로 동작하는지를 보기 위해서 남겨두었겠지만, 나중에 코드를 읽는 사람에게는 왠지 모를 암박감을 주게된다. 마치 지워서는 안되는 코드처럼 생각되기 때문이다. 따라서, 이런 코드들은 왜 이렇게 되었는지 이해하려는 노력이 추가되며, 항상 그 이유는 모호하기 마련이다. 따라서, 쓸데없는 고민을 방지하기 위해서도 삭제하는 것이 바람직하다. 코드는 될 수 있으면 깔끔하게 유지되도록 해야한다.

[코드에 계층구조 도입하기]

보통의 경우 대부분의 소프트웨어는 크게 3개의 계층으로 구분할 수 있다. 전체 시스템 차원에서, 그리고, 하위의 서브 시스템, 모듈이나 컴포넌트 수준에서도 3개의 계층은 일반적으로 가져간다. 첫 번째 계층은 외부에서 호출되는 부분을 담당하게 되며, 하위 계층의 전체적인 수행을 연결하게 된다. 두 번째 계층은 실제로 해야할 일을 구현하는 부분이며, 마지막 세 번째 계층은 하위 연결(포팅:Porting)을 위해서 필요하다. 따라서, 특정 시스템을 설계할 때 취할 수 있는 일반적인 접근 방법은 크게 3개의 계층으로 나누어서 그림을 보여준 후에, 각각의 계층에 대해서 다시 3개의 작은 계층으로 나누는 방식으로 진행될 수 있다.



하나의 모듈을 만들 때도 3계층으로 구성하는 것이 좋다. 첫 번째 계층은 모듈보다 상위 계층에 속하는 모듈들이 호출할 수 있는 인터페이스를 유지하기 위해서 필요하고, 두 번째 계층은 실제로 모듈에서 해야 할 일을 정의하며, 세 번째 계층은 외부의 하위 계층과의 연결에 사용된다. 이를 파일 내부에도 적용해 볼 수 있으며, 외부로 들어나는 인터페이스와 외부에 의존적인 인터페이스, 그리고, 실제로 파일 내부에서 사용되어 작업을 처리하는 함수들로 구성할 수 있을 것이다. 외부로 드러나는 인터페이스와 외부에 의존적인 인터페이스들은 모아서 관리하는 것이, 나중에 새로운 시스템이나 소프트웨어로 포팅(Porting)하기 쉽게 만들어 준다. 물론, 이런 계층화는 오버헤드를 감수하면서 진행되는 것이 일반적이다.

```
#ifdef __XXX_H__
#define EXT
#else
#define EXT extern
#endif

EXT void XXX_XXX( void );
EXT void YYY_YYY( void );
EXT int ZZZ; /* 변수를 외부에 공개하는 것은 좋지 않지만, 예를 위해서 남겨둔다. */
...
#undef EXT
```

위와 같은 코드는 헤더 파일에 외부로 보여줄 함수와 변수를 선언하는 방법이다. 이때, 소유권을 명시적으로 밝히는 효과를 가지기 위해서 "extern"을 매크로를 선언해서 만들었으며, 함수도 마찬가지로 선언되도록 했다. 이런 형태의 헤더 파일을 만들어서, 각각의 모듈에 템플릿 형태를 사용하면, 구현에 편의를 제공받을 수 있을 것이다.

```
#ifdef __cplusplus
extern "C" {
#endif

... /* Body of Header */

#ifndef __cplusplus
}
#endif
```

C++와 C를 같이 사용하기 위해서는 위와 같은 코드도 필요하다. 따라서, 앞에서 정의한 것과 위의 코드에서 정의한 것을 하나의 파일로 만들어서 템플릿으로 유지한다면 조금 더 도움이 될 것이다.

```
#ifndef __XXX_H__
#define __XXX_H__  
  
#ifdef __cplusplus
extern "C" {  
#endif  
  
#ifdef __EXTERNAL__
#define EXT extern
#else
#define EXT
#endif  
  
EXT void XXX_XXX( void );
EXT void YYY_YYY( void );
EXT int Z;  
  
#undef __EXTERNAL__
... /* Other Body of Header */...  
  
#ifdef __cplusplus
}  
#endif  
  
#endif /* End of #ifndef */
```

[하나의 파일에는 하나의 외부 공개 함수만 두기]

구현 파일을 만드는 방식에는 많은 방법이 있지만, 관리를 편하게 하기 위해서는 외부에 공개되는 함수 하나에 대해서 하나의 파일을 만드는 것이 도움이 될 때가 있다. 즉, 그 함수를 외부의 인터페이스로 제공하는 경우, 해당 함수의 이름을 파일의 이름으로 그대로 명시하고, 내부의 구현에는 그 함수의 구현에 필요한 것들만으로 채우는 것이다.

예를 들어, "allocate_memory_buffer()"라는 함수가 있다고 하자. 이 함수를 구현하는 파일의 이름은 "allocate_memory_buffer.c"가 될 것이다. 내부에는 이 함수를 구현하기 위한 작은 함수들의 모임으로 코드를 구성한다. 작은 함수들의 길이는 대부분 10에서 30라인(혹은, 최대 100라인 이하, 200라인 이상이면 문제가 발생할 가능성이 높다.) 정도로 구성하고, 외부로 공개되는 함수는 이들 작은 함수들의 조합으로 구현한다.

이렇게 해서 얻는 효과는 생각보다 크다. 즉, 우리가 어떤 함수의 문제를 찾을 때, 코드를 검색하는 방법으로 찾기도 하겠지만(주로, 편집기의 기능을 이용해서), 어떤 파일 만을 수정 대상으로 할지가 명확해진다. 그리고, 그것과 관련이 없는 코드들은 다른 파일로 분리해야 한다는 점이다. 즉, 응집성이 높은 코드가 만들어질 수 있다는 것이다.

파일의 길이가 너무 짧다고 고민할 필요가 없다. 그리고, 파일의 개수가 늘어난다고 해서 코드의 길이가 늘어나거나 오류가 늘어나는 것도 아니다. 오히려, 블록화되고 작은 함수들로 나누어진 코드들이 관리하기 쉽고 이해하기도 편하다. 만약, 다른 함수들을 필요로하고 그것이 공통된 요구라면, 그런 함수들로만

구성된 하위 디렉토리를 만들 수 있다. 이렇게 해서 만들어진 파일들로 구성된 것이 모듈이 되고, 서브 시스템을 계층적으로 구성하도록 만들어 준다.

만약, 다루고 있는 자료구조가 있다면, 그것도 하나의 파일에 같이 묶어두는 것이 좋다. 객체지향 언어에서 얻을 수 있는 캡슐화(Encapsulation)과 같은 것이 의미가 있다면, 데이터와 그것을 다루는 함수는 같이 정의되어 있어야 한다. 물론, 데이터를 사용하는 곳은 여러곳이 있을 수 있다. 따라서, 데이터의 상태 변화를 일으키거나, 데이터의 입출력을 담당하는 것들만을 일단은 모아서 관리해야 할 것이다. 데이터 자체를 외부에서 직접적으로 접근하는 것은 대부분의 경우 올바른 선택이 아니다.

[함수의 길이를 짧게 만드는 방법]

보통의 경우 코딩을 하다보면 이런 저런 생각과 처리해야 할 일들로 인해서 함수는 길어지기 마련이다. 함수는 최소 실행의 단위가 됨과 더불어 해야하는 일의 관점에서 추상화의 단위가 되기도 하기에, 함수를 최소한으로 유지하는 일은 중요한 기본기에 해당한다. 추상화란 실제 내부 구현에 대한 간략화라는 관점에서, 함수의 이름이 의미하는 바를 정확히 구현해 내야 한다. 커지는 함수의 길이를 짧게하는 방법은 함수 내부의 일 처리 단위를 분리된 작은 함수로 만드는 것이다.

```
void function( int a, int b, int c, int d ) {
    ...
    { /* function_a */
        ...
    }
    { /* function_b */
        ...
        { /* function_b_01 */
            ...
        }
    }
    { /* function_c */
        ...
    }
    ...
}
```

위의 코드는 길어지는 함수를 어떻게 나눌 것인가에 대한 간단한 예이다. 함수내의 각각의 논리적인 블록들은 자신만의 변수와 처리를 가질 수 있으며, 순차적인 처리를 하거나 혹은 병렬적인 처리로 구분될 수 있다. 이때, 각각의 논리적인 블록들은 하나의 함수로 분리될 수 있으며, 각각의 함수를 위한 파라미터는 전체를 감싸는 함수에서 사용하는 내부 변수들과 관련이 있다.

```
if( ( a < MAX ) && ( b > MIN ) && (( c % EVEN) == 0 ))
```

```
if( isMeetTheCondition( a, b, c ))
```

물론, 논리적인 블록들만 함수의 대상이 되는 것은 아니다. "if()"문과 같이 복잡한 조건을 가지거나, 반복문(Loop)문과 같은 경우에도 함수의 대상이 될 수 있다. 복잡한 논리식을 이름이 붙은 함수로 대체하면,

코드는 더 읽기 쉬운 경향을 가지게 된다. 반복문은 반복적인 처리를 나타내는 함수 이름으로 대체될 수 있다. 내부적으로 조건이나 반복을 깊이 생각해 보지 않더라도, 적절한 함수의 이름을 사용한다면, 충분히 읽기 좋은 코드를 만들 수 있게되는 것이다.

최적화를 중요하게 생각한다면, 반복문 내에서의 함수 호출을 자제해야 한다. 이때는 그냥 반복문 전체를 포함한 것을 함수로 만드는 거이 좋을 것이다. 함수의 호출 오버헤드(Overhead)는 크지 않지만, 자주 호출하는 것은 바람직하지 않다. 그리고, 이렇게 만든다고 코드의 구조까지 망가지는 것은 아니다.

```
for( int i = MIN_INT; i <= MAX_INT; i++ ) {
    do_something();
}
```

```
int do_something( void ) {
    for ( int i = MIN_INT; i <= MAX_INT; i++ ) {
        .... /* task done by do_something() */
    }
    return result;
}
```

코드가 알기 쉬운 구조를 띠는 것은 사람에게만 중요한 것이 아니라, 컴파일러에도 좋은 효과를 불러일으키게 된다. 즉, 최적화를 시도하는 컴파일러도 최적화 해야 할 부분이 줄어들기 때문에, 더 좋은 최적화를 할 가능성이 높다. 나중에 반복문에 대한 풀어헤침(Unrolling)을 통해서 변화를 줄 경우도 가정한다면, 반복문의 실행은 최적화에 영향을 많이 주는 부분이기에 최대한 짧은 반복과 메모리 접근을 막아주는 것이 좋자. 레지스터(Register)를 활용하기도 함수화 시키는 것이 더 유리할 수 있다.

"switch()"문과 같이 각각의 케이스(Case)마다 처리해야 할 들이 많아지는 경우에도 함수화는 이해를 높이는 수단으로 활용할 수 있다. 즉, 각각의 케이스에서 실행해야 할 부분들을 함수로 만드는 것이다. 기본적으로 "switch()"를 사용하는 것이 코드의 중복을 높일 수 있다는 점에서 권장하지는 않지만, 길어지는 케이스별 분기 실행에 대해서는 함수화를 할 필요가 있다. 전체 케이스를 묶어서 다시 함수로 정의 할 수도 있을 것이다.

```
switch( a ) {
    case 'A':
        function_a();
        break;
    case 'B':
        function_b();
        break;
    ...
    default:
        ...
}
```

초기 개발에서 전역변수를 많이 사용하는 경우, 해당 전역변수를 사용하는 논리적인 블록들을 함수로 만들어서 전역 변수에 대한 의존성을 분리하는 것이 가능하다. 전역변수의 의존성이 분리되면 코드는 더 안정적으로 동작하게 되며, 전역변수를 접근하는 부분은 분리되어 인터페이스로 유지되는 경향이 생긴다. 따라서, 전역변수의 자료구조가 변경 되더라도, 그외의 코드들은 변경되지 않을 가능성이 높다. 외부

에 의존하고 있는 부분들을 직접적으로 접근하는 것보다, 이런 식으로 분리된 함수로 처리하는 것이 향후 확장성에도 영향을 주게 된다.

함수를 작게 만드는 것은 프로그램의 "이해"를 높이는 동시에, 버그를 특정 코드에 한정(지역화) 시키도록 유지하는데 도움을 준다. 될 수 있으면 작은 코드를 많이 만드는 것이 핵심이다. 긴 코드는 누구나 만들 수 있지만, 짧고 복제되지 않게 유지하는 일은 경험있는 개발자의 필수다.

짧은 코드는 당연히 하는 일도 명확히 구분될 것이다. 하는 일이 명확한 함수는 이름 만들기도 쉽다. "xxx_and_yyy()"라는 함수 이름은 제대로 역할을 정의하지 못한 경우에 흔히 발생한다. "common"이나 "normal", "utility"와 같은 너무 일반적인 이름도 좋은 이름이 아니다. 함수는 명확히 정의된 한가지 역할만 최선을 다해서 수행해야 한다.

[함수의 명확성을 높이는 역할 분리]

함수는 크게 두 종류의 일을 처리한다. 하나는 데이터를 변경하는 일이고, 하나는 데이터를 얻어오는 일이다. 이 두 가지를 섞어놓고 구현하면 함수는 데이터를 변경도 하지만, 데이터를 얻어오는 일도 가능하게 된다. 이때, 데이터를 변경하는 것은 시스템의 상태 변경을 일으키게 되며, 데이터를 얻어오는 것은 시스템의 상태는 변경하지 않는다. 따라서, 두 가지를 함께 사용하면 그 만큼 함수는 복잡해지게 된다. 이것을 방지하는 방법은 함수를 명확하게 데이터를 처리하는 것과 데이터를 얻는 것으로 각각 나누어서 구현하는 것이다. 즉, 한번에 할 수 있는 일이지만 나누어서 구현하는 것이다.

```
#include <stdio.h>
#include <stdlib.h>

#define MAXIMUM_ENTRIES 40
static unsigned int Data[MAXIMUM_ENTRIES];

void init_data_entries(void) {
    for (unsigned int i = 0; i < MAXIMUM_ENTRIES; i++) {
        Data[i] = 0;
    }
}

unsigned int make_fibonacci_sequence(unsigned int xth) {
    /* Sanity Check */
    if (xth >= MAXIMUM_ENTRIES) {
        printf("Cannot calculate fibonacci sequence number for %d\n", xth);
        exit(1);
    }
    /* 1st and 2nd */
    if ((xth <= 0) || (xth == 1)) {
        Data[xth] = 1;
        return 1;
    }
    /* Others */
    if ((Data[xth - 2] != 0) && (Data[xth - 1] != 0)) {
        Data[xth] = Data[xth - 2] + Data[xth - 1];
        return Data[xth];
    } else {
        Data[xth] = make_fibonacci_sequence(xth - 2)
                    + make_fibonacci_sequence(xth - 1);
    }
}
```

```

        return Data[xth];
    }
}

unsigned int get_data_entries(unsigned int xth) {
    return Data[xth];
}

void print_data_entries(void) {
    for (unsigned int i = 0; (i < MAXIMUM_ENTRIES) && (Data[i] != 0); i++) {
        printf("%dth Fibonacci Number : %d\n", i + 1, Data[i]);
    }
}

int main(void) {
    puts("Data manipulation");
    init_data_entries();
    make_fibonacci_sequence( MAXIMUM_ENTRIES - 1 );
    print_data_entries();

    return EXIT_SUCCESS;
}

```

위의 코드는 Fibonacci 숫자를 구하는 것을 재귀적인 호출을 사용하지만, 만약 기존에 계산된 결과가 있을 때는 그것을 이용하도록 수정해서 만든 것이다. 여기서 "init_data_entries()", "make_fibonacci_sequence()"등의 함수는 데이터의 변경에 관여하는 함수들이다. 재귀적인 함수인 "make_fibonacci_sequence()"함수는 예외로 하더라도, 데이터의 변경만을 다루는 함수는 복귀값을 가지지 않도록 만들었다. 나머지 "get_data_entries()"와 "print_data_entries()"와 같은 경우에는 데이터의 변경에는 관여하지 않으며, 단순히 데이터를 얻어오는(출력하는) 일만 담당했다. 즉, 시스템의 상태 변경에는 관여하지 않는다. "get_data_entries()"와 같은 함수는 복귀값도 가질 수 있도록 만들었다.

일반적으로 시스템의 변경을 일으키는 일과 시스템의 정보를 알아오는 일은 분리되는 것이 옳은 선택이다. 즉, 같이 묶어놓으면 함수간에 순서에 대한 의존성이 생길 가능성이 높다. 즉, 변경과 동시에 정보를 얻어오는 것은 반드시 변경의 결과가 업데이트 된 결과만을 얻어오지만, 분리할 경우에는 변경과 상관없이 정보를 얻어 올 수 있다. 함수가 잘 정의된 하나의 일만 해야한다고 가정 했을 때, 이렇게 분리하는 것이 역시 하나의 일만 함수가 더 잘 담당할 수 있도록 만들어준다. 따라서, 함수를 이해하는 것도 좀더 쉬워진다.

[참고] 일반적인 컴퓨터에서 Fibonacci 숫자를 재귀호출만 사용해서 구현할 경우에는 40정도가 넘어가면, 구하는 시간이 길어질 수 있다. 또한, 50쯤에 도달하면 정수의 범위를 초과하는 값이 만들어지게 되어 잘못된 값이 나올 수 있다. 범위를 초과하는 값을 다루기 위해서는 배열과 같은 것을 사용해서 연속적으로 값을 채워넣을 수 있도록 미리 준비해야 하며, 큰 값에 대해서는 나누어서 처리하는 함수가 필요하다.

앞에서는 일단 구현하는데 집중을 했다면, 다음의 구현은 함수들을 데이터를 조작하는 것과 그렇지 않은 것으로 나누어 서로 간섭이 최소화 되도록 만들었다. 코딩은 한번 만들고 다시 보는 것이 아니라, 계속 개선해야 한다는 점에서 약간 기능을 보완했다. 즉, 커맨드 라인에서 입력을 줄 수 있는 방식으로 간단히 수정했다. 함수의 선언도 전달되는 파라미터들의 변경이 없다는 것을 명확히 보여주기 위해서 "const"를 사용했다. 파일의 내부에서만 사용되는 함수라는 것을 나타내기 위해서 "static"으로 정의했으며, 사용되지 않았던 함수는 삭제해서 코드를 더 깔끔하게 만들었다.

```

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAXIMUM_ENTRIES 47

static unsigned int Data[MAXIMUM_ENTRIES];

static void init_data_entries(void) {
    for (unsigned int i = 0; i < MAXIMUM_ENTRIES; i++) {
        Data[i] = 0;
    }
}

static inline void check_boundary(const unsigned int xth) {
    if (xth >= MAXIMUM_ENTRIES) {
        printf("Cannot calculate fibonacci sequence number for more than : %d\n",
MAXIMUM_ENTRIES);
        exit(1);
    }
    return;
}

static unsigned int make_fibonacci_sequence(const unsigned int xth) {
    /* Sanity Check */
    check_boundary(xth);

    /* 1st and 2nd */
    if ((xth <= 0) || (xth == 1)) {
        Data[xth] = 1;
        return 1;
    }
    /* Others */
    if ((Data[xth - 2] != 0) && (Data[xth - 1] != 0)) {
        Data[xth] = Data[xth - 2] + Data[xth - 1];
    } else {
        Data[xth] = make_fibonacci_sequence(xth - 2)
                    + make_fibonacci_sequence(xth - 1);
    }
    return Data[xth];
}

static inline bool isEndofData(const unsigned int i, const unsigned int x) {
    return ((i < MAXIMUM_ENTRIES - 1) && (x != 0));
}

void print_data_entries(void) {
    for (unsigned int i = 0; isEndofData(i, Data[i]); i++) {
        printf("%3dth Fibonacci Number : %10d\n", i + 1, Data[i]);
    }
}

```

```

static unsigned int make_argument(int argc, const char *argv[]) {
    if (argc > 1) {
        return atoi(argv[1]); /* 주의!!! atoi() 함수는 오류를 유발할 가능성도 있다. */
    }
    return ( MAXIMUM_ENTRIES - 1);
}

int main(int argc, const char *argv[]) {
    unsigned int xth = 0;

    puts("Data manipulation");
    init_data_entries();
    xth = make_argument(argc, argv);
    make_fibonacci_sequence(xth);
    print_data_entries();

    return EXIT_SUCCESS;
}

```

정리한 이후의 코드도 사실 아직 개선할 부분이 남았다. 함수들 간에 눈으로 드러나는 의존성은 없지만(함수를 호출하는 입장에서 함수의 정의만 본다면), 내부적으로는 "Data[]"와 같은 자료구조를 공유하고 있으며, "MAXIMUM_ENTRIES"와 같은 값을 여러 함수에서 공통으로 사용하고 있다. 따라서, 이런 것들이 변경이 발생할 경우 영향을 받을 수 있는 범위는 늘어나게 된다. 좀더 의존성을 줄이기 위해서는 함수들이 내부적으로 사용하는 자료구조에 대해서도 의존성이 없도록 만들어야 할 것이다.

이것은 사실 함수의 파라미터 갯수를 늘리는 방법을 해결할 수 있다. 즉, 함수가 처리해야 할 정보를 외부에서 직접 제공하는 것이다. 물론, 그렇게 할 경우 함수는 점점 더 복잡해져 갈 것이다. 따라서, 구현하는 사람의 입장에서는 얼마나 복잡하게 만들지에 대해서 어느 정도 타협할 지점이 필요하다. 이때 사용할 수 있는 방법은 물리적으로 함수 파라미터의 수 4개 이하로 유지하는 것이다. 4개까지는 스택(Stack)을 이용하지 않고 레지스터를 이용해서 파라미터를 전달할 수 있기 때문이다. 하지만, 가능한 함수의 파라미터 개수는 적은게 더 좋다.

추가적으로 수정한 부분은 작은 오류들이며, 주로 경계값에서 발생하는 잘못된 결과들을 정리했다. 버그가 주로 발생하는 부분은 함수와 함수의 호출 사이와 함수 호출 시에 발생하는 파라미터의 경계값들이 원인이 되는 경우가 많다. 따라서, 프로그램을 테스트하는 입장에서는 그런 부분을 신경써서 확인해 주어야 한다. 그리고, 프로그램 편집기에서 제공하는 기능을 이용해서, 특정 코딩 스타일로 전체 코드의 포맷을 변경했다. 이렇게 해주는 이유는 프로그램을 읽는 사람이 일관된 코딩 스타일로 읽을 수 있도록 가독성을 높여준 것이다.

[“goto”문의 올바른 사용]

일반적으로 "goto"와 같이 제어(Control:명령 실행 순서)를 크게 변동시키는 코드는 좋지 못하다고 말한다. 제어가 움직인다는 말은 프로그램이 실행할 다음 명령어를 찾는 것(IP:instruction Pointer)이 크게 값이 달라진다는 뜻이며, 예상하지 못한 상황을 맞을 수 있다는 것이다. 따라서, "goto"문은 될 수 있으면 사용하지 않는 것이 현명하다. 하지만, 그렇다고 사용하는 것이 전혀 도움이 되지 않는 것은 아니다. 예를 들어, 오류가 발생한 상황에서 빨리 빠져나오는 방법으로 사용할 수도 있다. 따라서, 몇 가지 주의 사항만 잘 지킨다면 유용한 도구로 활용해 볼 충분한 가치도 있다.

label :

...

```
goto label;
...
```

일반적으로 "goto"로 제어를 옮기는 위치는 "goto"문보다 나중에 두는 편이 좋다. 따라서, 위의 코드는 잘못된 "goto"의 사용이라고 볼 수 있다. 이와 같은 경우를 대체 할 수 있는 방법은 조건문이나 반복문 등이 될 수 있다. 문제가 발생한 상황에서 주로 빠져나오는 용도인 경우에는, 제어 구조를 단순화하는데 효과적인 경우로 한정해 볼 수 있다.

```
...
if (isValid()) {
    ...
} else {
    ...
    goto label; /* "Forward" goto를 사용했다. */
}
...
label:
...
```

위와 같은 경우는 "goto" 레이블이 "goto"문 이후에 나타난다. 코드를 읽는 경우 좀 더 자연스런 경우라고 볼 수 있다. 자연스럽기에 오류를 유발할 가능성도 줄어든다. 위의 코드는 오류를 확인하고 신속히 오류 처리로 제어를 옮기기 위해서 "goto"를 사용했다.

"goto"의 레이블 위치도 중요하다. 일반적으로 하나의 함수를 벗어나는 레이블을 사용해선 안된다. 가능한 "goto"문과 가까운 위치에 두어야 할 것이다. 같은 논리적인 블록이나 하나 정도 벗어난 수준으로 사용을 한정하는 것이 제어를 옮기는데 따른 혼란을 방지해 줄 수 있을 것이다. 급격한 제어의 변경은 생각의 흐름을 끊어지게 만들어 연속된 상황에서 처리해야 할 일을 놓일 가능성이 많다. 나중에 코드를 수정해야 할 일이 있다면, 이것은 버그를 발생시킬 가능성도 높아지게 된다.

[참조(Reference) 및 호출(Call)]

코드의 의존성이 높아지는 이유는 두 가지가 있다. 하나는 다른 함수를 사용하는데 들어가는 연결이며, 다른 하나는 자료구조를 직접적으로 다루는데 따른 것이다. 즉, 둘 다 내부적인 변경에 대해서 사용자 측의 코드 변경을 동반할 가능성이 높다. 또 다른 한 가지의 이유를 더 추가하자면, 함수나 자료구조에 대한 책임과 역할이 명확하지 않다는 점이다. 즉, 어떤 모듈이(혹은, 함수가) 어떤 일을 해주어야 하는지 명확하지 않아서, 서로가 서로를 의존하도록 만드는 경우에 해당한다.

```
/* main.c */
#include <stdio.h>
extern int fibonacci(int xth);

void print_result(int result) {
    printf("%d\n", result);
}

int main(int argc, const char* argv[]) {
    fibonacci();
    return 0;
}

/* fibonacci.c */
```

```

extern void print_result(int result);

int fibonacci(int xth) {
    int temp;

    if ((xth <= 0) || (xth == 1)) {
        return 1;
    }
    temp = fibonacci(xth - 1) + fibonacci(xth - 2);
    print_result(temp);
    return temp;
}

```

만약, 위와 같이 "main.c"와 "fibonacci.c"가 각각 코딩되어 있다고 생각하자. 코드에서 보듯이, "main.c"는 "fibonacci.c"의 "fibonacci()" 함수를 사용하는 의존성을 가지고, 다시 "fibonacci.c"는 "main.c"에서 가지고 있는 "print_result()"를 사용하는 의존성을 가지고 있다. 즉, 두 개의 파일로 분리되어 있지만, 사실 상은 하나의 파일과 같다느 것이다. 하나의 파일을 수정하면 다른 파일이 영향을 받을 확률이 높다. 이렇게 간단히 코드를 만들어서 보여주면 당연히 이렇게 코딩하는 경우는 없다고 이야기 할 것이다. 하지만, 코드가 조금만 복잡해져도 이런 일은 빈번하게 발생한다. 예를 들어, "A.c --> B.c --> C.c --> D.c ... --> A.c"와 같은 순환적인 의존성이 생성되는 경우가 많다.

이와 같이 순환적인 의존성을 끊는 것은 생각보다 간단하다. 크게 두 가지 방법을 활용할 수 있는데, 첫 번째가 함수의 위치를 옮기는 것이다. 즉, 순환의 고리상 마지막에 위치하는 함수를 사용되는 위치로 옮겨놓는 것이다. 물론, 이름도 같이 변경해 주는 것이 좋다. 예를 들어, 모듈의 이름을 함수의 이름 앞에 사용하는 것은 함수의 위치를 명확히 해 줄 수 있다는 점에서 도움이 된다(임베디드 시스템에서 주로 사용). 순환 고리상 자료구조에 의존적인 코드라 그곳에 밖에 위치할 수 없다고 이야기할 경우도 있을 것이다. 이때는 사용할 자료구조와 함수를 같이 넘겨서 콜백(Callback)함수로 만들어 줄 수도 있다.

```

/* main.c */
#include <stdio.h>
#include <stdlib.h>
extern int fibonacci(int xth);
extern void fibonacci_set_callback(void (*callback)(int result));

void print_result(const int result) {
    printf("%d\n", result);
}

int main(int argc, char* argv[]) {
    fibonacci_set_callback(print_result);
    fibonacci(10);
    return EXIT_SUCCESS;
}

/* fibonacci.c */
typedef void (*Callback)(int result);
Callback print_result_callback;

extern void fibonacci_set_callback(Callback func) {
    print_result_callback = func;
}

```

```

}

int fibonacci(int xth) {
    int temp;

    if ((xth <= 0) || (xth == 1)) {
        print_result_callback(1);
        return 1;
    }
    temp = fibonacci(xth - 1) + fibonacci(xth - 2);
    print_result_callback(temp);
    return temp;
}

```

위의 코드는 앞에서 생겼던 순환적인 의존성의 문제를 "main.c"가 "fibonacci.c"를 사용하는 일방향 (Unidirectional)의 의존 관계로 바꿔준 경우다. 콜백과 같은 것을 사용했다고, 순환적인 의존관계를 가질 것일고 생각한다면 잘못된 생각이다. 호출되는 콜백 함수는 설정된 값을 이용해서 호출될 뿐이며, 콜백을 사용하는 측에서는 어떤 함수가 호출될지 알지 못한다. 따라서, 콜백을 정의할 때 만들어진 함수의 타입과 파라미터의 타입만 보고 호출가능 여부를 판단할 뿐이다. 따라서, 호출은 발생하지만, 호출되는 콜백을 저장한 부분은 코드의 내부에 정의된 변수를 이용하기에 변수의 값 변화와 같이 생각할 수 있다.

함수 자체를 옮기는 것은 쉽다. 위와 같은 코드의 경우에는 "fibonacci.c"로 "print_result()"함수를 옮겨주는 거이 가능하다. 하지만, 이렇게 한다면, "fibonacci.c" 파일의 역할 범위를 초과한 것이다. 따라서, 차라리 새로운 파일로 만들어 "print"기능을 분리 시키는 것이 더 좋은 선택일 것이다. 파일의 갯수는 늘어나지만, 하나의 파일이 책임져야 하는 범위는 좀더 명확하게 드러나게 되어, 변경해야 할 부분은 축소할 수 있다.

```

/* main.c */
#include <stdio.h>
#include <stdlib.h>

extern int fibonacci(const int xth);
extern void fibonacci_set_callback(void (*callback)(int result));
extern void print_result(const int result);

int main(int argc, const char* argv[]) {
    fibonacci_set_callback(print_result);
    fibonacci(10);
    return EXIT_SUCCESS;
}

/* print_result.c */
#include <stdio.h>

void print_result(const int result) {
    printf("%d\n", result);
}

/* fibonacci.c */
typedef void (*Callback)(int result);

```

```

Callback print_result_callback;

extern void fibonacci_set_callback(const Callback func) {
    print_result_callback = func;
}

int fibonacci(const int xth) {
    int temp;

    if ((xth <= 0) || (xth == 1)) {
        print_result_callback(1);
        return 1;
    }
    temp = fibonacci(xth - 1) + fibonacci(xth - 2);
    print_result_callback(temp);
    return temp;
}

```

위의 코드는 그 전에 보여준 코드와 큰 차이점은 없지만, 사용하는 파라미터들의 변경이 발생하지 않는다는 것을 호출하는 측에 알려주기 위해서 "const"를 붙여주었다. 이렇게 하는 이유는 호출 이후에 변경되는 값을 가지는 경우, 일명 "부대효과(Side Effect)"를 예방함과 더불어, 호출되는 함수에 대한 인터페이스 규약을 한정짓도록 만들어준다. 따라서, 좀 더 안전하게 함수를 사용할 수 있게된다. 그리고, 추가적으로 결과 출력 함수를 관련된 파일(새로운 파일로 정의해서)로 옮겨 주었다.

의존관계 중에서 자료구조의 의존관계나 사용되는 자료의 의존관계를 정리하기 위해서는 좀 더 주의가 필요하다. 자료구조에 대한 의존성은 자료구조를 다루는 전담 함수들을 만들어서 처리하면 되지만, 만약, 긴 함수의 체인(Chain)을 따라 내려가야 하는 파라미터를 이용한다면, 그것이 과연 필요한지도 생각해야 할 것이다. 예를 들어, IP(Internet Protocol) 패킷과 같은 것을 처리해야하는 경우, 각각의 계층을 거치면서 해주어야 할 일이 다르다. 그리고, 자주 데이터를 옮기는 것도 메모리 복사를 발생시켜 처리속도를 낮춘다.

이런 상황이라면, 패킷을 할당받을 경우 속도 향상을 위해서 CPU 구조에 맞는 바이트(Byte)단위의 정렬(Align)을 만들어야 하고, 잊은 데이터의 복사를 줄이기 위해 포인터를 이용해서 함수를 호출 하게 될 것이다. 데이터의 이동 대신에 패킷을 나타내는 자료구조를 사용하고, 그 자료구조에 실제 데이터를 저장하는 공간에 대한 포인터를 유지할 수 있다.

[역할과 책임(Role & Responsibility)의 구현]

코드를 역할과 책임의 관점에서 구조적으로 만들기 위해서, "Print"에 관련된 함수를 분리된 파일로 만들었다. "main.c"파일은 더 깔끔하게 정리되었고, 필요한 코드 이외에는 보이지 않는다. 의존관계는 "main.c -> PrintResult.c"와 "main.c -> Fibonacci.c"로 나누어졌으며, 순환 의존관계나 역호출(Reverse Call)은 보이지 않는다.

```

/* main.c */
#include <stdio.h>
#include <stdlib.h>

#include "Fibonacci.h"
#include "PrintResult.h"

int main(int argc, const char* argv[]) {

```

```

        fibonacci_set_callback(print_result);
        fibonacci(10);
        return EXIT_SUCCESS;
    }

/* PrintResult.h */
#ifndef PRINTRESULT_H_
#define PRINTRESULT_H_
extern void print_result(const int result);
#endif /* PRINTRESULT_H_ */

/* PrintResult.c */
#include <stdio.h>
#include "PrintResult.h"

void print_result(const int result) {
    printf("%d\n", result);
}

/* Fibonacci.h */
#ifndef FIBONACCI_H_
#define FIBONACCI_H_

extern int fibonacci(const int xth);
extern void fibonacci_set_callback(void (*callback)(int result));

#endif /* FIBONACCI_H_ */

/* Fibonacci.c */
#include "Fibonacci.h"

typedef void (*Callback)(int result);
Callback print_result_callback;

void fibonacci_set_callback(const Callback func) {
    print_result_callback = func;
}

int fibonacci(const int xth) {
    int temp;

    if ((xth <= 0) || (xth == 1)) {
        print_result_callback(1);
        return 1;
    }
    temp = fibonacci(xth - 1) + fibonacci(xth - 2);
    print_result_callback(temp);
    return temp;
}

```

함수 별로 새로운 파일로 만들고 외부 인터페이스를 담당하는 헤더 파일을 제공하는 것은 모듈화를 높이는 방법으로 주로 사용된다. 모듈화 된 코드는 재활용이 쉬우며, 효과적으로 코드에 내재하는 복잡성과 의존성을 다루는 방법이다. 큰 프로그램을 만들더라도 이와 같은 기본적인 규칙을 지켜야만 코드의 가독성과 변경가능성, 유지보수성 등을 높일 수 있다. 코드가 고치기 어려워지는 이유는 이런 모듈화라는 기본원리를 지키지 않고 개발되기 때문이다. 모듈이라고 이야기 하는 대부분의 코드들이 모듈이 아닌 이유는 서로 의존성으로 강하게 묶여서 별도로 관리할 수 없기 때문이다.

[자료구조 숨기기]

자료구조에 대한 의존성은 자료구조를 숨기는 것으로 가능하다. 즉, 일체의 자료구조를 외부로 알리지 않는 것이다. 예를 들어, 선언된 자료구조의 타입 만을 외부에 공개하고, 해당 자료구조를 사용하는 함수들을 만들어서 외부에 제공하는 것이다. 나중에 자료구조의 수정이나 함수의 변경에 대해서 영향을 받지 않는 클라이언트(Client: 자료구조를 및 제공되는 함수를 사용하는 측) 코드를 만들 수 있도록 한다.

```
#include <stdio.h>
#include <stdlib.h>

#include "HiddenDataStructure.h"

/* 클라이언트 코드 */
int main(void) {
    HIDDENSTRUCT *myinfo = NULL;

    puts("====Hidden Data Structure Test!!!====");
    if ((myinfo = createPersonalInfo("SH Kwon", 20, MALE)) == NULL) {
        printf("Cannot set personal information!!!\n");
        return EXIT_FAILURE;
    }

    printPersonalInfo(myinfo);

    if (setPersonalInfo(myinfo, "MJ Yoon", 20, FEMALE) == false) {
        printf("Cannot set personal Information!!!\n");
        freePersonalInfo(myinfo);
        return EXIT_FAILURE;
    }

    printf("======\n");
    printf("Name : %s\nAge : %d\n", getPersonName(myinfo),
           getPersonAge(myinfo));
    if (getPersonSex(myinfo) == MALE) {
        printf("Sex : Male\n");
    } else {
        printf("Sex : Female\n");
    }
    printf("======\n");

    freePersonalInfo(myinfo);

    return EXIT_SUCCESS;
}
```

"main.c" 파일에서는 자신이 처리하고자 하는 작업을 위해서 필요한 데이터의 타입 만을 알 뿐이지, 그것이 정확히 어떻게 구현되었는지에 관심이 없다. 단순히 API로 정의된 인터페이스를 이용해서 원하는 동작을 수행하기만 하면 된다. 코드에서 보듯이 "HIDDENSTRUCT"라는 구조체가 나중에 변경되더라도, "main.c" 파일은 변경 될 가능성이 줄어든다. 자료구조에 대한 모든 처리는 다른 곳에 정의된 함수에 의해서 수행될 것이기 때문이다.

```
#ifndef HIDDENDATASTRUCTURE_H_
#define HIDDENDATASTRUCTURE_H_

#include <stdbool.h>

typedef enum { MALE = 0, FEMALE = 1 } SEX;
typedef struct HiddenDataStructure HIDDENSTRUCT;

extern HIDDENSTRUCT *createPersonallInfo( char *name, unsigned int age, SEX sex );
extern char *getPersonName( HIDDENSTRUCT *info );
extern int getPersonAge( HIDDENSTRUCT *info );
extern SEX getPersonSex( HIDDENSTRUCT *info );
extern bool setPersonallInfo( HIDDENSTRUCT *info, char *name, unsigned int age, SEX sex );
extern void printPersonallInfo( HIDDENSTRUCT *info );
void freePersonallInfo(HIDDENSTRUCT *info);

#endif /* HIDDENDATASTRUCTURE_H_ */
```

위의 헤더 파일은 외부에 보여주기 위한 것이다. 따라서, 이 헤더 파일을 사용하는 측에서는 필요한 API와 전달될 파라미터의 타입 등의 정보만 알 수 있다. 내부적인 자료구조에 대한 것은 이곳에 정의되어 있지 않다. 나중에 코드를 배포할 일이 생긴다면, 필요한 코드들을 라이브러리로 만들고, 사용을 위해서 헤더 파일만 제공하면 될 것이다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#include "HiddenDataStructure.h"

typedef struct HiddenDataStructure {
    char *name;
    unsigned int age;
    SEX sex;
} HIDDENSTRUCT;

HIDDENSTRUCT *createPersonallInfo(char *name, unsigned int age, SEX sex) {
    HIDDENSTRUCT *personallInfo;

    if ((personallInfo = (HIDDENSTRUCT *) malloc(sizeof(HIDDENSTRUCT))) == NULL) {
        return NULL;
    }
```

```

if ((personalInfo->name = (char*) malloc(strlen(name) + 1)) == NULL) {
    free(personalInfo->name);
    personalInfo->name = NULL;
    return NULL;
}
strncpy(personalInfo->name, name, strlen(name) + 1);
personalInfo->age = age;
personalInfo->sex = sex;
return personalInfo;
}

char *getPersonName(HIDDENSTRUCT *info) {
    char *name;

    if (info->name == NULL) {
        return NULL;
    }

    if ((name = (char*) malloc(strlen(info->name) + 1)) == NULL) {
        return NULL;
    }

    strncpy(name, info->name, strlen(info->name) + 1);
    return name;
}

int getPersonAge(HIDDENSTRUCT *info) {
    return info->age;
}

SEX getPersonSex(HIDDENSTRUCT *info) {
    return info->sex;
}

bool setPersonalInfo(HIDDENSTRUCT *info, char *name, unsigned int age, SEX sex) {
    if (info->name != NULL) {
        free(info->name);
    }
    if ((info->name = (char*) malloc(strlen(name) + 1)) == NULL) {
        return false;
    }
    strncpy(info->name, name, strlen(name) + 1);
    info->age = age;
    info->sex = sex;
    return true;
}

void printPersonalInfo(HIDDENSTRUCT *info) {
    printf("=====\\n");
    printf("Name : %s\\n", info->name);
    printf("Age : %d\\n", info->age);
    switch (info->sex) {

```

```

case MALE:
    printf("Sex : Male\n");
    break;
case FEMALE:
    printf("Sex : Female\n");
    break;
default:
    break;
}
printf("=====\\n");
return;
}

void freePersonalInfo(HIDDENSTRUCT *info) {
    if (info->name != NULL) {
        free(info->name);
    }
    free(info);
    return;
}

```

클라이언트 코드를 만든다면, 서비스를 제공하는 측의 내부 구현에 의존적이지 않도록 만들어야 한다. 즉, 서비스의 내부적인 구현 변경의 영향도를 최소화해야 한다. SOLID 원칙 중에서 OCP(Open Close Principle)는 “확장에 대해서는 열려있고, 변경에 대해서는 닫혀있게 만들라”는 것은 이와 같은 의존성을 단방향으로(Client->Server 혹은 Caller->Callee)만 유지하며, 인터페이스를 사용해야만 구현이 가능하다.

[NULL Object를 활용한 검사 코드의 제거]

어떤 함수를 호출할 때 만약 그 함수가 자료구조의 포인터를 복귀값으로 돌려준다면, 우리는 항상 그 함수의 복귀값이 "Null"값이 아닌지 확인해서 처리를 해야한다. 사실 이것은 함수를 호출하는 측에 호출당하는 함수가 해야 할 오류 처리를 넘기는 일이며, 호출하는 측에서는 "Null"인 경우와 그렇지 않은 경우 각각에 대해서 나누어 처리해야 한다는 의무를 지우게된다. 만약, 적절한 처리를 하지 못하게 된다면, 시스템은 "Null'참조에 의해서 오류를 발생시킬 것이다.

이러한 경우, "Null"인 상황에서도 원하는 자료구조를 복귀값으로 돌려주지만, 특별한 용도의 자료구조를 생성해서 넘겨주는 방법으로 해결할 수 있다. 이럴 때 사용되는 자료구조를 널 객체("Null Object")라고 부르며, 객체지향 프로그램에서는 흔히 사용하는 방법이다. 즉, 아무런 일도 해주지 않는 "Default" 자료구조를 하나 만들어서, 복귀 값이 "Null"인 경우에 전달하는 것이다.

```

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

typedef struct Personal_Information PERSONAL_INFORMATION;
struct Personal_Information {
    char *name;
    unsigned int age;
    bool married;
    void ( *print_personal_information )( PERSONAL_INFORMATION *info );
};

PERSONAL_INFORMATION *create_null_info() {
    PERSONAL_INFORMATION *info = (PERSONAL_INFORMATION *)malloc(sizeof(PERSONAL_INFORMATION));
    info->name = NULL;
    info->age = 0;
    info->married = false;
    info->print_personal_information = NULL;
    return info;
}

void print_personal_information(PERSONAL_INFORMATION *info) {
    if (info->print_personal_information != NULL) {
        info->print_personal_information(info);
    }
}

int main() {
    PERSONAL_INFORMATION *info = create_null_info();
    print_personal_information(info);
    free(info);
    return 0;
}

```

위의 코드는 자료구조의 내부정보와 그것을 다루는(Handler) 함수를 같이 선언해서, 외부에서 직접적으로 자료구조에 대한 접근을 막기 위해서 만들었다. 즉, 자료구조에 대한 접근은 자료구조 내에 정의된 함수로만 한정하고, 초기화 시에 그것들을 설정해 줄 수 있도록 하면 된다. 이 자료구조를 사용하는 측에서는 내부구조를 알지 못해도 원하는 일을 처리할 수 있는 방법을 제공받을 수 있다. 나중에 자료구조의 변경이 발생하더라도, 이 자료구조를 사용하는 측에서는 코드를 변경하지 않고도 변화에 대응할 수 있게된다.

```
void my_print_personal_information( PERSONAL_INFORMATION *info ) {
    printf( "Name : %s\n", info->name );
    printf( "Age : %d\n", info->age );
    printf( "Married : %s\n", (( info->married ) ? ("Yes") : ("No")));
}
```

```
void null_print_personal_information( PERSONAL_INFORMATION *info ) {
    /* Nothing to do */
    printf( "Null Information!!!\n");
}
```

```
PERSONAL_INFORMATION NullInfo = { "No Name", 0, false,
                                null_print_personal_information };
```

널 객체를 정의된 자료구조의 형(Type)과 동일하게 생성해 주었다. 다른점이라면, 호출되는 함수에서는 아무런 일도 처리하지 않는다는 점이다("null_print_personal_information()"). 나중에 “NULL”상황을 처리하기 위해서 널 객체의 주소를 이용하기 위해서 이렇게 정의해 주었다.

```
PERSONAL_INFORMATION *allocate_structure(const char *name, const int age,
                                         const bool married) {
    PERSONAL_INFORMATION *templInfo;

    /* Allocate required memory */
    if ((templInfo = (PERSONAL_INFORMATION *) malloc(
        sizeof(PERSONAL_INFORMATION))) == NULL) {
        printf("Cannot allocate required memory : error 01!!!");
        return &NullInfo;
    }
    if ((templInfo->name = (char*) malloc(strlen(name) + 1)) == NULL) {
        printf("Cannot allocate required memory : error 02!!!");
        free(templInfo);
        return &NullInfo;
    }
    strncpy(templInfo->name, name, strlen(name) + 1);
    templInfo->age = age;
    templInfo->married = married;
    templInfo->print_personal_information = my_print_personal_information;
    return templInfo;
    //return &NullInfo;
}
```

자료구조를 생성하는 코드를 보여준다. 실제로 프로그램이 동작하는 중간에는 메모리 할당과 같은 호출이 실패할 가능성이 있으며(대부분의 경우에는 거의 발생하지 않지만), 그것으로 인해서 항상 메모리가 제대로 할당되었는지를 확인하는 과정이 메모리 할당 함수와 쌍으로 쓰이는 경향이 있다. 여기서는 메모리 할당이 실패했을 경우, 그것을 처리하기 위한 "Null Object"의 주소를 돌려주는 것으로 대체 했다. 만약, "Null Object"의 동작을 확인하고 싶다면, 가장 마지막 라인의 "//"를 제거하고, 그 위 라인에 "//"으로 주석한 후에 실행해 보면 될 것이다.

```
int main(void) {
    puts("Null Object Test 01");
    puts("=====");
    PERSONAL_INFORMATION *myinfo;

    /* Allocate structure */
    myinfo = allocate_structure("SH Kwon", 44, true);
    /* Print information */
    myinfo->print_personal_information(myinfo);
    puts("=====");

    return EXIT_SUCCESS;
}
```

위의 코드에서 보듯이 "allocate_structure()"라는 함수는 자료구조를 생성해서 포인터를 전달해 준다. 만약, 널 객체가 없었다면, 이곳에서는 복귀값이 "Null"인 경우와 그렇지 않은 경우에 대해서 각각 나누어서 처리해 주어야 했을 것이다. 하지만, 이제는 그런 것을 따지지 않고 일관된 처리할 수 있게 되었다.

한 곳에서만 해주면 될 오류에 관련된 처리를 시스템 전체로 전파시킬 필요는 없다는 생각에서 널 객체와 같은 것을 도입했는데, 이런 식으로 코드를 작성한다면 호출하는 측의 코드는 좀 더 깔끔하고 일관되게 보일 것이다. 함수를 이용하는 측에서는 항상 함수가 기본적인(Default) 동작을 준비하고 있다는 생각에 안심하고 해당 함수를 호출할 수 있을 것이며, 함수와 함수내에서 사용하는 자료구조에 대한 독립성도 얻게된다(파일을 분리해서 정리한 후 Binary Image와 헤더 파일만 제공할 수 있음). 될 수 있으면 자료의 공개를 꺼리고, 사용하는 측에서 쉽게 이용할 수 있는 방법을 제공하는 것이, 코드간에 인터페이스(interface) 역할을 할 수 있는 함수를 잘 만드는 방법이다.

```
/* main.c */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

#include "PersonalInformation.h"

int main(void) {
    puts("Null Object Test 01");
    puts("=====");
    PERSONAL_INFORMATION *myinfo;

    /* Allocate structure */
    myinfo = allocate_information_structure("SH Kwon", 44, true);

    /* Print information */
```

```

        print_information(myinfo);
        puts("=====");
        return EXIT_SUCCESS;
    }

/* PersonalInformation.h */
#ifndef PERSONALINFORMATION_H_
#define PERSONALINFORMATION_H_

typedef struct Personal_Information PERSONAL_INFORMATION;

#ifdef __cplusplus
extern "C" {
#endif

extern PERSONAL_INFORMATION *allocate_information_structure(const char *name,
                const int age, const bool married);
extern void print_information(PERSONAL_INFORMATION *info);

#ifdef __cplusplus
}
#endif

#endif /* PERSONALINFORMATION_H_ */

/* personalInformation.c */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

#include "PersonalInformation.h"

struct Personal_Information {
    char *name;
    unsigned int age;
    bool married;
    void (*print_personal_information)(PERSONAL_INFORMATION *info);
};

static void my_print_personal_information(PERSONAL_INFORMATION *info) {
    printf("Name : %s\n", info->name);
    printf("Age : %d\n", info->age);
    printf("Married : %s\n", ((info->married) ? ("Yes") : ("No")));
}

static void null_print_personal_information(PERSONAL_INFORMATION *info) {
    /* Nothing to do */
    printf("Null Information!!!\n");
}

PERSONAL_INFORMATION NullInfo = { "No Name", 0, false,

```

```

        null_print_personal_information };

PERSONAL_INFORMATION *allocate_information_structure(const char *name,
    const int age, const bool married) {
    PERSONAL_INFORMATION *tempInfo;

/* Allocate required memory */
if ((tempInfo = (PERSONAL_INFORMATION *) malloc(
        sizeof(PERSONAL_INFORMATION))) == NULL) {
    printf("Cannot allocate required memory : error 01!!!");
    return &NullInfo;
}
if ((tempInfo->name = (char*) malloc(strlen(name) + 1)) == NULL) {
    printf("Cannot allocate required memory : error 02!!!");
    free(tempInfo);
    return &NullInfo;
}
strncpy(tempInfo->name, name, strlen(name) + 1);
tempInfo->age = age;
tempInfo->married = married;
tempInfo->print_personal_information = my_print_personal_information;
return tempInfo;
//return &NullInfo;
}

void print_information(PERSONAL_INFORMATION *info) {
    info->print_personal_information(info);
}

```

위의 코드는 파일을 분리시키고, "main.c"파일의 작성자가 제공되는 헤더 파일만을 이용해서 코드를 작성했다. "main.c"파일의 작성자는 자료구조의 내부를 보지 않아도 충분히 원하는 외부로 공개되는 API들을 활용해서 적절하게 일을 처리할 수 있으며, 메모리 할당 실패에 대해서도 신경쓸 필요가 없다. 따라서, 함수의 사용에 대해서 좀 더 호출자의 의존성이 줄어들게 되었다. 추가적으로 자료구조에 구체적인 일을 처리하는 함수들은 파일 범위에서 한정적으로 사용될 수 있도록 "static"으로 선언되었다. 직접적인 외부에서 접근할 필요가 없기 때문이다.

[리플렉션(Reflection)을 이용한 자신의 자료구조 접근]

리플렉션이란 컴퓨터 프로그램이 자기 자신의 구조(Structure)나 행위(Behavior)를 실행 중에 조사하거나, 수정할 수 있는 능력(Ability)을 말한다. Java나 .Net과 같은 곳에서는 동적으로 변화에 대처하기 위해서 사용할 수 있다. C언어에서는 기본적으로 리플렉션은 지원되지 않지만, 그것을 구현하기 위한 약간의 방법 정도만 여기서 간단히 보기로 하겠다.

```

#ifndef REFLECTOR_H_
#define REFLECTOR_H_

typedef struct reflector REFLECTOR;
struct reflector {
    char name[100];
    unsigned int age;
    bool married;

```

```

REFLECTOR *(*set_name)(REFLECTOR *info, const char *name);
REFLECTOR *(*set_age)(REFLECTOR *info, const unsigned int age);
REFLECTOR *(*set_marital_status)(REFLECTOR *info, const bool married);
REFLECTOR *(*get_name)(REFLECTOR *info, char **name);
REFLECTOR *(*get_age)(REFLECTOR *info, unsigned int *age);
REFLECTOR *(*get_marital_status)(REFLECTOR *info, bool *married);
};

extern REFLECTOR personal_info;

#ifndef __cplusplus
extern "C" {
#endif

extern void print_personal_info(REFLECTOR *info);

#ifndef __cplusplus
}
#endif
#endif /* REFLECTOR_H_ */

```

위에서는 객체지향 언어에서 사용하는 클래스 대신에 구조체("struct")를 이용해서 데이터와 데이터를 처리하는 함수들의 포인터를 선언해 주었다. 헤더 파일에 이와 같이 구체적인 정보를 명시하는 것은 좋은 일은 아니지만, 여기서 그냥 이렇게 남겨두도록 했다. 나중에 이 부분을 다른 함수를 이용해서 가릴 수 있지만, 여기서는 "main.c"파일에서 직접 함수에 접근할 수 있도록 구조체를 상세히 정의해 주었다.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#include "Reflector.h"

static REFLECTOR *set_name(REFLECTOR *info, const char *name) {
    strncpy(info->name, name, strlen(name) + 1);
    return info;
}

static REFLECTOR *set_age(REFLECTOR *info, const unsigned int age) {
    info->age = age;
    return info;
}

static REFLECTOR *set_marital_status(REFLECTOR *info, const bool married) {
    info->married = married;
    return info;
}

static REFLECTOR *get_name(REFLECTOR *info, char **name) {
    if ((*name = (char *) malloc(strlen(info->name) + 1)) == NULL) {
        printf("Cannot allocated memory!!!\n");
        return info;
    }
}

```

```

    }
    strncpy(*name, info->name, strlen(info->name) + 1);
    return info;
}

static REFLECTOR *get_age(REFLECTOR *info, unsigned int *age) {
    *age = info->age;
    return info;
}

static REFLECTOR *get_marital_status(REFLECTOR *info, bool *married) {
    *married = info->married;
    return info;
}
REFLECTOR personal_info = { .set_name = set_name, .set_age = set_age,
                            .set_marital_status = set_marital_status, .get_name = get_name,
                            .get_age = get_age, .get_marital_status = get_marital_status };

void print_personal_info(REFLECTOR *info) {
    printf("=====\\n");
    printf("Name : %s\\n", info->name);
    printf("Age : %d\\n", info->age);
    printf("Married : %s\\n", (info->married) ? "Married" : "Not Married");
    printf("=====\\n");
}

```

여기서 사용한 "struct"의 각 필드에 대한 초기화 방법은 "C99" 표준에 새롭게 추가된 부분이다. 즉, 이런 식으로 개별 필드를 표시하고 초기화 시켜줄 수 있다. 예전에는 각각의 필드의 순서를 기억할 필요가 있었지만, 이제는 해당 필드의 이름을 직접 사용해서 초기화가 가능하다.

각각의 함수들은 일종의 접근자(Accessor) 역할을 하는 "get, set"들로 정의했다. 한 가지 특별한 부분은 자기가 다루는 자료구조의 포인터를 복귀값으로 넘겨준다는 것이다. 포인터에 대해서 변경이 발생할 경우에는 "Call-by-Reference"를 구현하기 위해서 "***"을 사용했다("get_name()"). 실패한 경우라고 해도, 일단은 원래의 구조체에 대한 포인터를 복귀값으로 전달하도록 만들었다.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#include "Reflector.h"

void print_information(const char *name, const unsigned int age,
                      const bool married) {
    printf("=====\\n");
    printf("Name : %s\\n", name);
    printf("Age : %d\\n", age);
    printf("Married : %s\\n", (married) ? "Married" : "Not Married");
    printf("=====\\n");
}

```

```

int main(void) {
    puts("ReflectionExample 01");

    (((personal_info.set_name(&personal_info, "SH Kwon))->set_age(
        &personal_info, 44))->set_marital_status(&personal_info, true));

    print_personal_info(&personal_info);

    char *name = NULL;
    unsigned int age = 0;
    bool married = false;

    (((personal_info.get_name(&personal_info, &name))->get_age(&personal_info,
        &age))->get_marital_status(&personal_info, &married));

    print_information(name, age, married);

    return EXIT_SUCCESS;
}

```

정의된 자료구조를 이용해서 함수를 호출한 후에, 복귀 값을 이용해서 다시 다른 내부에 정의된 함수들을 호출하는 형식으로 만들어진 것을 확인할 수 있을 것이다. 대략적으로 보이게 C언어 문법같이 보이지는 않으며, 이해가 잘 안갈 수도 있을 것이다. 이렇게 코딩하는 것이 좋은 습관은 아니지만, 일단은 이렇게 사용할 수 있다는 정도만 익혀도 될 것이라고 생각한다. 좋은 코드는 이해가 잘되는 코드라는 점을 항상 기억하기 바란다.

예제에서 보듯이 자기 자신에 대한 정보를 조회하고, 포인터에 새로운 함수를 설정해서 동적으로 다른 동작을 실행할 수 있도록 만들 수 있다는 점도 알 수 있을 것이다. 나중에 객체지향 언어와 같은 것을 배우게 될 때, 리플렉션(Reflection)이 무엇인지 이미 경험해 보았다는 수준에서 이 코드를 읽었으면 한다.

[스택(Stack)의 활용]

스택은 일반적으로 많이 사용되는 자료구조이다. 들어간 순서와 반대로 나오도록 만들기 위해서 흔히 사용된다. LIFO(Last In First Out)는 알고리즘과 같은 것을 구현할 때 흔히 사용되기에, 여기서 잠시 그 구현을 구조체를 이용해서 만들어 보았다. 나중에 입력의 종류를 변경하기 위해서 실제 데이터를 가지는 부분에 대한 타입을 새로 정의해서 활용하면 될 것이다.

```

#include <stdio.h>
#include <stdlib.h>

#define MAX_STACK_SIZE 10
typedef struct stack STACK;
struct stack {
    char *inputs[MAX_STACK_SIZE];
    int top;
    void (*push)(STACK *stack, char *input);
    char *(*pop)(STACK *stack);
};

```

구조체는 데이터를 저장하기 위한 포인터의 배열("inputs")과, 스택의 저장위치를 기록하기 위한 "top", 스택에 데이터를 저장하는 "push()" 함수, 저장된 데이터를 가져오는 "pop()" 함수의 포인터들로 구성된

다. 여기서는 구현의 단순화를 위해서 입력되는 데이터는 문자열에 대한 포인터로 한정했다.

```
static void push(STACK *stack, char *input) {
    if (stack->top >= MAX_STACK_SIZE) {
        printf("Cannot push into the stack!!!\n");
        return;
    }
    stack->inputs[stack->top] = input;
    stack->top++;
    printf("The input : %s\n", input);
    return;
}

static char *pop(STACK *stack) {
    if (stack->top <= 0) {
        printf("Cannot pop the empty stack!!!\n");
        return NULL;
    }
    stack->top--;
    return stack->inputs[stack->top];
}
```

스택의 "top"은 항상 저장될 위치를 가르키지만, 크기가 다 차면 저장할 수 있는 공간보다 하나 더 큰 값을 가지도록 만들었다. 나중에 "pop()"함수에서 이를 고려해 주어야 할 것이다. 더 이상 저장할 공간이 없으면, 오류 메시지를 출력하고 바로 복귀하도록 만들었다.

"pop()"함수에서는 더 이상 저장된 데이터가 없으면, 오류 메시지를 출력하고 "NULL"값을 돌려주도록 한다. 이미 "top"이 저장된 데이터의 위치보다 하나 더 큰 위치를 가리키고 있기에, 데이터를 가져오기 위해서는 미리 "top"의 위치를 변경해야 한다는 점에 유의해야 할 것이다. 추가적으로 "top"이 이미 "0"의 값을 가지면, 이미 이야기 했듯이 더 이상 가져올 데이터가 없다는 뜻이 된다.

```
int main(void) {
    puts("Stack Example 01");
    STACK stack = { { }, 0, push, pop };
    char *push[] = { "0", "1", "2", "3", "4", "5", "6", "7", "8", "9" };

    for (int i = 0; i < MAX_STACK_SIZE; i++) {
        stack.push(&stack, push[i]);
    }
    for (int i = 0; i < MAX_STACK_SIZE; i++) {
        printf("The popped : %s\n", stack.pop(&stack));
    }

    stack.push(&stack, "Hello, World!!!");
    stack.push(&stack, "Hello, World!!!");
    printf("The popped : %s\n", stack.pop(&stack));
    printf("The popped : %s\n", stack.pop(&stack));
    return EXIT_SUCCESS;
}
```

예제에서는 10개의 숫자를 나타내는 문자열을 스택에 저장 했다가 다시 가져오는 것을 만들어 보았다. 추가적으로 다른 문자열에 대한 것도 저장하고 가져오는 것을 테스트 했다. 아래는 예제를 실행한 결과이다.

[결과]

Stack Example 01

```
The input : 0
The input : 1
The input : 2
The input : 3
The input : 4
The input : 5
The input : 6
The input : 7
The input : 8
The input : 9
The popped : 9
The popped : 8
The popped : 7
The popped : 6
The popped : 5
The popped : 4
The popped : 3
The popped : 2
The popped : 1
The popped : 0
The input : Hello, World!!!
The input : Hello, World!!!
The popped : Hello, World!!!
The popped : Hello, World!!!
```

[큐(Queue)의 활용]

스택(Stack)과 더불어 가장 많이 사용되는 것이 큐이다. 특히, 한정된 공간을 가진 큐는 이용되는 빈도가 많으며, 그런 큐에 자료를 넣거나 빼는 것은 빈번하게 발생한다. 그 중에서도 큐가 마치 원을 따라 움직이는 것처럼 보인다고 해서 이를 붙여진 순환 큐(Circular Queue:원형 큐)는 활용 빈도가 높다.

순환 큐를 구현하기 위해서는 어디에 추가하고, 어디서 가져올 것 인가를 정해야 한다. 그리고, 한 가지 주의할 부분은 큐의 빈 상태와 가득 찬 상태를 구분하기 어렵다는 것이다. 따라서, 이를 위해서는 여분의 공간이나 코드의 추가가 불가피하다.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct element {
    char *message;
} ELEMENT;

ELEMENT nullElement = { "Null Element" }; /* NULL element for replacing NULL pointer */

#define MAX_QUEUE_SIZE 5
```

```
typedef struct queue QUEUE;
struct queue {
    ELEMENT *elements[MAX_QUEUE_SIZE];
    int head;
    int tail;
    void (*add)(QUEUE *queue, ELEMENT *element);
    ELEMENT *(*get)(QUEUE *queue);
};
```

큐에 일반적인 자료구조를 저장하기 위해서 메시지를 구조체에 담을 수 있도록 만들었다. "nullElement"는 "NULL"조건에서 오류를 검출하는 코드를 만들 필요가 없도록 추가되었다. 큐에는 자료가 추가되는 위치를 가리키는 "head"와 자료를 가져올 위치를 가리키는 "tail"을 정의했다. "add()"와 "get()"은 큐에 자료를 추가하거나, 빼기위해서 사용한다.

```
void add(QUEUE *queue, ELEMENT *element) {
    int next = queue->head + 1;
    /* Wrap around */
    if (next >= MAX_QUEUE_SIZE) {
        next = 0;
    }
    if (next == queue->tail) {
        printf("Queue is full!!!\n");
        return;
    }
    queue->elements[queue->head] = element;
    printf("The Queue Head : %d\n", queue->head);
    queue->head = next;
    return;
}
```

큐에 자료구조를 넣기 위해서는, 먼저 자료구조를 넣어야 할 빈 공간을 가르키고 있는 "head"를 증가시켜서, 그것이 "tail"과 같은 값인지 확인해야 한다. 즉, 그 상태에서는 이미 큐에 빈 공간이 없다고 생각할 수 있다. 하지만, 한 가지 문제는 처음에 빈 상태도 그와 같다는 것이다. 따라서, 여기서는 인위적으로 "head"와 "tail"사이에는 항상 1개의 빈 공간을 만들어서, 그것을 통해서 큐가 가득찼다는 것을 알려주도록 했다.

```
ELEMENT *get(QUEUE *queue) {
    ELEMENT *element;

    if (queue->head == queue->tail) {
        printf("Queue is empty!!!\n");
        return &nullElement;
    }
    element = queue->elements[queue->tail];
    printf("The Queue Tail : %d\n", queue->tail);
    queue->elements[queue->tail] = NULL;
    int next = queue->tail + 1;
    if (next >= MAX_QUEUE_SIZE) {
        next = 0;
    }
    queue->tail = next;
```

```

    return element;
}

```

큐에서 데이터를 가져오기 위해서는 큐가 비지 않아야 한다. 따라서, 큐의 "head"와 "tail"이 같은 값을 가지는지 부터 확인한다. 비었을 경우에는 이미 만들어둔 "null" 값을 가지는 자료구조의 포인터를 리턴 한다. 자료를 큐에 끄집어낸 후에는 다음 자료를 가리키는 위치로 "tail"을 옮겨준다.

```
QUEUE queue = { .elements = { }, .head = 0, .tail = 0, .add = add, .get = get };
```

```

int main(void) {
    puts("Queue Example 01");

    ELEMENT element00 = { "Element 00" };
    ELEMENT element01 = { "Element 01" };
    ELEMENT element02 = { "Element 02" };
    ELEMENT element03 = { "Element 03" };
    ELEMENT element04 = { "Element 04" };
    ELEMENT element05 = { "Element 05" };
    ELEMENT element06 = { "Element 06" };
    ELEMENT element07 = { "Element 07" };

    queue.add(&queue, &element00);
    queue.add(&queue, &element01);
    queue.add(&queue, &element02);
    queue.add(&queue, &element03);
    queue.add(&queue, &element04); /* Queue is full!!! */
    queue.add(&queue, &element05); /* Queue is full!!! */
    printf("The element : %s\n", (queue.get(&queue))->message);
    printf("The element : %s\n", (queue.get(&queue))->message);
    queue.add(&queue, &element04);
    queue.add(&queue, &element05);

    printf("The element : %s\n", (queue.get(&queue))->message);
    queue.add(&queue, &element06);
    printf("The element : %s\n", (queue.get(&queue))->message);
    queue.add(&queue, &element07);
    printf("The element : %s\n", (queue.get(&queue))->message);
    printf("The element : %s\n", (queue.get(&queue))->message); /* Queue empty!!! */
    /* The element : Null Element */
    return EXIT_SUCCESS;
}

```

구현의 편의를 위해서 큐는 정적인 자료구조로 정의했다. 그리고, 몇 개까지 큐에 받아들이는지를 확인하기 위해서 큐에 들어가야할 자료구조 들의 개수를 여러 개 만들었다. 아래의 실행한 결과에서 볼 수 있듯이 큐의 크기는 5이지만, 실제로 큐에 들어갈 수 있는 개수는 4개 까지다. 즉, 1개의 공간은 큐가 비었는지, 혹은 가득 찾는지를 나타내기 위해서 사용한다.

결과에서는 4개 까지 큐를 채운 후에는 큐에서 데이터를 끄집어내면서 출력과 삽입을 여러 번 수행했다. 이미 빈 큐에서 데이터를 끄집어 내려고 하면, "NULL" 대신에 "nullElement"의 포인터를 돌려준다는 것을 알 수 있다. Null을 대표하는 자료구조는 일반적으로 삽입에 사용한 자료구조와 동일하기에, 복귀 값을 받는 측에서는 그 내용을 보지 않는한, 어떤 것을 받았는지 구분하지 못한다. 따라서, "NULL"을 따로 검사할 필요는 없다. 여기서는 출력을 만들어주는 값을 사용했지만, 아무런 출력이 나오지 않도록만 들어도 상관없다.

[결과]

```

Queue Example 01
The Queue Head : 0
The Queue Head : 1
The Queue Head : 2
The Queue Head : 3
Queue is full!!!
Queue is full!!!
The Queue Tail : 0
The element : Element 00
The Queue Tail : 1
The element : Element 01
The Queue Head : 4
The Queue Head : 0
The Queue Tail : 2
The element : Element 02
The Queue Head : 1
The Queue Tail : 3
The element : Element 03
The Queue Head : 2
The Queue Tail : 4
The element : Element 04
The Queue Tail : 0
The element : Element 05
The Queue Tail : 1
The element : Element 06
The Queue Tail : 2
The element : Element 07
Queue is empty!!!
The element : Null Element

```

[포인터를 이용한 스택의 구현]

앞에서는 배열을 사용해서 스택을 구현 했지만, 일반적으로 가변 길이를 가질 경우에는 입력에 대해서 미리 정의된 값을 사용하기 어려울 수 있다. 이 경우에는 포인터를 이용해서 처리할 수 있다. 기본적으로 메모리 할당에 관련된 것은 다루지 않고, 단순히 연결된 형태의 리스트(List)만 관리하도록 만들었다.

```

#include <stdio.h>
#include <stdlib.h>

typedef struct entry ENTRY;
struct entry {
    char *data;
    ENTRY *next;
    ENTRY *prev;
}

```

```
};

typedef struct stack STACK;
struct stack {
    ENTRY *head;
};

void push(STACK *stack, ENTRY *entry) {
    if ((stack == NULL) || (entry == NULL)) {
        return;
    }
    /* Empty? */
    if (stack->head == NULL) {
        stack->head = entry;
        printf("Push : value : %s\n", stack->head->data);
        return;
    }
    entry->prev = stack->head;
    entry->next = NULL;
    stack->head->next = entry;
    stack->head = entry;
    printf("Push : value : %s\n", stack->head->data);
    return;
}
```

```
ENTRY *pop(STACK *stack) {
    ENTRY *temp;
    if ((stack == NULL) || (stack->head == NULL)) {
        printf("Wrong or empty stack!!!\n");
        return NULL;
    }
    temp = stack->head;
    if (stack->head->prev == NULL) {
        stack->head = NULL;
        return temp;
    }
    stack->head->prev->next = NULL;
    stack->head = stack->head->prev;
    temp->next = NULL;
    temp->prev = NULL;
    return temp;
}
```

```
int main(void) {
    puts("StackPointerExample01");
    STACK stack = { NULL };
    ENTRY entry01 = { "SH Kwon", NULL, NULL };
    ENTRY entry02 = { "MJ Yoon", NULL, NULL };
    ENTRY entry03 = { "JY Kwon", NULL, NULL };
    ENTRY entry04 = { "JW Kwon", NULL, NULL };
    ENTRY *entry;
    push(&stack, &entry01);
```

```

push(&stack, &entry02);
push(&stack, &entry03);
push(&stack, &entry04);

/* Make sure that stack is ok? */
entry = stack.head;
while (entry != NULL) {
    printf("The List : %s\n", entry->data);
    entry = entry->prev;
}

entry = pop(&stack);
while (entry != NULL) {
    printf("The Value : %s\n", entry->data);
    entry = pop(&stack);
}
return EXIT_SUCCESS;
}

```

[결과]

```

StackPointerExample01
Push : value : SH Kwon
Push : value : MJ Yoon
Push : value : JY Kwon
Push : value : JW Kwon
The List : JW Kwon
The List : JY Kwon
The List : MJ Yoon
The List : SH Kwon
The Value : JW Kwon
The Value : JY Kwon
The Value : MJ Yoon
The Value : SH Kwon
Wrong or empty stack!!!

```

포인터를 이용하면 자료구조의 형에 의존하지 않는 일반화된 알고리즘을 구현할 수 있다. 하지만, 그만큼 더 주의해서 코드를 구현해야 하는 것이 일반적이다. 될 수 있으면 구체적인 자료구조에 맞게 먼저 구현하고, 나중에 일반화 시키는 것이 좋은 구현 전략이다. 즉, 일반화 시켜야 할 이유가 발생할 경우에 다양한 자료구조의 형을 만족 시키도록 구현하는 것이다.

앞의 코드에서 보듯이, 스택이 저장할 수 있는 원소의 한계값이 사라졌기에, 관리할 수 있는 카운터(Counter)를 만들어 넣을 수도 있을 것이다. 간혹 잘못 사용하는 경우 무한정 늘어나는 스택을 볼 수 있기에, 적절한 값으로 한계치를 설정하는 것도 좋은 방법이다.

[포인터를 이용한 큐의 구현]

아래의 코드는 큐를 배열 대신에 포인터를 이용해서 구현한 것을 보여 준다. 메모리 할당은 처리하지 않았으며, 큐를 사용하는 측에서 자신이 추가하고 싶은 것을 이미 만들었다고(할당해서 초기화 시켰다고) 가정한다.

```
#include <stdio.h>
#include <stdlib.h>
```

```

typedef struct entry ENTRY;
struct entry {
    ENTRY *next;
    ENTRY *prev;
    char *data;
};

typedef struct queue QUEUE;
struct queue {
    ENTRY *head;
    ENTRY *tail;
};

void enqueue(QUEUE *queue, ENTRY *entry) {
    if ((queue == NULL) || (entry == NULL)) {
        printf("Wrong queue or entry for queuing!!!\n");
        return;
    }
    /* Check empty */
    if ((queue->tail == NULL) && (queue->head == NULL)) {
        queue->tail = entry;
        queue->head = entry;
        return;
    }
    /* If not empty */
    entry->prev = queue->head;
    queue->head->next = entry;
    queue->head = entry;
    return;
}

ENTRY nullEntry = { NULL, NULL, "Null Entry" };

ENTRY *dequeue(QUEUE *queue) {
    ENTRY *entry;

    if (queue == NULL) {
        printf("Wrong queue!!!\n");
        return &nullEntry;
    }
    if (queue->tail == NULL) {
        return &nullEntry;
    }
    entry = queue->tail;
    if (queue->tail->next != NULL) {
        queue->tail->next->prev = NULL;
    }
    queue->tail = queue->tail->next;
    if (queue->tail == NULL) {
        queue->head = NULL;
    }
    entry->next = NULL;
}

```

```

entry->prev = NULL;
return entry;
}

int main(void) {
    puts("Queue Pointer Example 01");

    QUEUE queue = { NULL, NULL };
    ENTRY entry01 = { NULL, NULL, "SH Kwon" };
    ENTRY entry02 = { NULL, NULL, "MJ Yoon" };
    ENTRY entry03 = { NULL, NULL, "JY Kwon" };
    ENTRY entry04 = { NULL, NULL, "JW Kwon" };
    enqueue(&queue, &entry01);
    enqueue(&queue, &entry02);
    enqueue(&queue, &entry03);
    enqueue(&queue, &entry04);

    printf("Queue value : %s\n", dequeue(&queue)->data);
    printf("Queue value : %s\n", dequeue(&queue)->data); /* Queue value : Null Entry */

    enqueue(&queue, &entry01);
    printf("Queue value : %s\n", dequeue(&queue)->data);
    enqueue(&queue, &entry02);
    printf("Queue value : %s\n", dequeue(&queue)->data);
    enqueue(&queue, &entry03);
    printf("Queue value : %s\n", dequeue(&queue)->data);
    enqueue(&queue, &entry04);
    printf("Queue value : %s\n", dequeue(&queue)->data);
    enqueue( NULL, &entry01);
    enqueue(&queue, NULL);
    enqueue( NULL, NULL);
    dequeue( NULL);

    return EXIT_SUCCESS;
}

```

[결과]

Queue Pointer Example 01
 Queue value : SH Kwon
 Queue value : MJ Yoon
 Queue value : JY Kwon
 Queue value : JW Kwon
 Queue value : Null Entry
 Queue value : SH Kwon
 Queue value : MJ Yoon
 Queue value : JY Kwon
 Queue value : JW Kwon
 Wrong queue or entry for queuing!!!
 Wrong queue or entry for queuing!!!

Wrong queue or entry for queuing!!!

Wrong queue!!!

포인터를 이용한 큐의 구현은 앞에서 본 배열을 이용한 순환 큐에서 필요했던 빈 공간을 사용할 필요가 없다. 또한, 큐 자체에서 더 추가할 공간이 있는지 확인해야 하는 부분도 사라졌다. 앞에서 본 스택과 마찬가지로 잘못 구현될 큐에서 대기하고 있는 원소들의 수가 지속적으로 증가할 가능성이 있기에, 한계값을 설정하는 것이 필요할 수도 있다. 여기서는 단순히 사용법 만을 보여주기 위한 것에 만족하도록 한다.

[포인터를 이용한 큐의 구현 개선]

이번에 보는 큐의 구현은 내부에 정의된 인라인 함수를 통해서 가독성(Readability)을 높여준 경우다. 코드를 읽는 사람의 입장에서는 좀 더 명료하게 구현된 코드가 하는 일을 알 수 있다.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct entry ENTRY;
struct entry {
    ENTRY *next;
    ENTRY *prev;
    char *data;
};

typedef struct queue QUEUE;
struct queue {
    ENTRY *head;
    ENTRY *tail;
};

static inline bool isQueueBad(QUEUE *queue) {
    return (queue == NULL);
}

static inline bool isEntryBad(ENTRY *entry) {
    return (entry == NULL);
}

static inline bool isEmpty(QUEUE *queue) {
    return ((queue->tail == NULL) || (queue->head == NULL));
}

static inline void setFirst(QUEUE *queue, ENTRY *entry) {
    queue->tail = entry;
    queue->head = entry;
    return;
}

static inline void insert(QUEUE *queue, ENTRY *entry) {
    entry->prev = queue->head;
    queue->head->next = entry;
}
```

```

queue->head = entry;
return;
}

static inline bool isMoreThan2Entry(QUEUE *queue) {
    return (queue->tail->next != NULL);
}
static inline void moveNext(QUEUE *queue) {
    queue->tail = queue->tail->next;
}

static inline void markEmpty(QUEUE *queue) {
    queue->tail = NULL;
    queue->head = NULL;
}

static inline void cleanEntry(ENTRY *entry) {
    entry->next = NULL;
    entry->prev = NULL;
    return;
}

static inline void cleanNextEntry(QUEUE *queue) {
    queue->tail->next->prev = NULL;
}

static inline ENTRY *saveTail(QUEUE *queue) {
    return queue->tail;
}

ENTRY nullEntry = { NULL, NULL, "Null Entry" };

void enqueue(QUEUE *queue, ENTRY *entry) {
    if (isQueueBad(queue) || isEntryBad(entry))
        return;
    if (isEmpty(queue)) {
        setFirst(queue, entry);
        return;
    }
    insert(queue, entry);
    return;
}

ENTRY *dequeue(QUEUE *queue) {
    ENTRY *entry = NULL;

    if (isQueueBad(queue))
        return &nullEntry;
    if (isEmpty(queue))
        return &nullEntry;
    if (isMoreThan2Entry(queue))
        cleanNextEntry(queue);
}

```

```

entry = saveTail(queue);
moveNext(queue);
if (isEmpty(queue)) {
    markEmpty(queue);
}
cleanEntry(entry);
return entry;
}

int main(void) {
    puts("Queue Pointer Example 01");

    QUEUE queue = { NULL, NULL };
    ENTRY entry01 = { NULL, NULL, "SH Kwon" };
    ENTRY entry02 = { NULL, NULL, "MJ Yoon" };
    ENTRY entry03 = { NULL, NULL, "JY Kwon" };
    ENTRY entry04 = { NULL, NULL, "JW Kwon" };

    enqueue(&queue, &entry01);
    enqueue(&queue, &entry02);
    enqueue(&queue, &entry03);
    enqueue(&queue, &entry04);

    printf("Queue value : %s\n", dequeue(&queue)->data);
    printf("Queue value : %s\n", dequeue(&queue)->data);

    enqueue(&queue, &entry01);
    printf("Queue value : %s\n", dequeue(&queue)->data);
    enqueue(&queue, &entry02);
    printf("Queue value : %s\n", dequeue(&queue)->data);
    enqueue(&queue, &entry03);
    printf("Queue value : %s\n", dequeue(&queue)->data);
    enqueue(&queue, &entry04);
    printf("Queue value : %s\n", dequeue(&queue)->data);

    enqueue( NULL, &entry01);
    enqueue(&queue, NULL);
    enqueue( NULL, NULL);
    dequeue( NULL);

    return EXIT_SUCCESS;
}

```

[결과]

Queue Pointer Example 01
 Queue value : SH Kwon
 Queue value : MJ Yoon
 Queue value : JY Kwon
 Queue value : JW Kwon

```
Queue value : Null Entry
Queue value : SH Kwon
Queue value : MJ Yoon
Queue value : JY Kwon
Queue value : JW Kwon
```

코드에서 보듯이 조건이나 함수내의 블록된 코드들을 별도의 인라인(Inline)함수로 뽑아냈다. 뽑아낸 인라인 함수들은 외부에서 접근할 필요가 없기에 “static”으로 정의 되었다. 코드는 좀 더 명확하게 이해하기 쉽게 변경되었지만, 함수의 호출이 늘어난 것을 확인할 수 있을 것이다. 물론, 이것은 나중에 컴파일러가 적절히 코드를 인라인 시켜줄 수 있기에 큰 오버헤드(Overhead)는 아니다. 하지만, 성능이 극한으로 필요한 경우에는 오버헤드로 사람들이 “오해”할 가능성이 있다.

함수를 작게 만드는 것에 대해서는 개발자들 사이에서도 서로 상반된 의견을 가지고 있는 부분이 있지만, 작게 만드는 것이 코드를 이해하기 쉽게 만든다는 점은 확실하다. 따라서, 될 수 있으면 작은 함수를 만드는 것을 우선해야 할 것이다. 나중에 성능이 잘 나지 않는 부분에 해당하는 함수들을 찾는다면, 그 부분만 나누어진 함수들을 합쳐도 상관없다. 더 성능이 필요하다면 해당 부분만 기계어(Assembly)로 구현할 수도 있을 것이다. 따라서, 일단은 무조건 함수를 짧게 만드는데 집중해 보는 것이 코드를 이해하기 쉽게 만드는 지름길이라고 생각해야 할 것이다.

[커맨드 라인(Command Line)의 해석]

커맨드 라인을 해석하는 것은 일반적으로 많은 프로그램들이 이미 해왔다. 특히, 콘솔로 명령어를 직접 입력하는 프로그램은 일반적으로 실행 옵션을 설정하기 위해서, 커맨드 라인을 많이 사용해 왔다. 이번에는 자주 사용되는 이러한 기능을 어떻게 처리하는지를 보여주도록 하겠다. C언어로 만들어진 프로그램을 실행하기 위해서는, 프로그램의 초기화를 담당하는 부분이 라이브러리 형태로 존재하며, 커맨드 라인에서 주어진 명령어를 기록해서 "main()" 함수로 전달해 주게된다. 커맨드 라인에 몇 개의 단어가 있는지를 표현하는 것이 "main()" 함수의 "argc"이며, 배열의 각각이 문자열을 가리키는 "argv"에 실행되는 프로그램의 이름과 나머지 주어진 명령어 문자열을 저장해서 전달해 준다.

이렇게 주어진 문자열들은 보통 “-”를 이용해서 옵션의 종류를 표현하며, 추가적인 부분이 필요하면 그 다음에 옵션에 대한 추가 정보를 표시하는 방법으로 주로 사용한다. 이는 일반적으로 Unix와 같은 시스템에서 주로 사용하던 방식이었으며, 일종의 전통과 같은 형태로 지금까지 유지되고 있다. 따라서, 각각의 옵션을 “-”로 시작하도록 예제에서도 사용했으며, 옵션의 종류를 사용하는 문자열과 옵션에 추가적인 정보를 주는 문자열은 이어서 나온다고 가정했다.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

void print_usage(void) {
    printf("=====\\n");
    printf("Usage:\\n");
    printf("CommandLineParsingExample [-x<name>] [-y<name>][-z<name>]\\n");
    printf("=====\\n");
    exit(1);
}

bool xOption = false;
bool yOption = false;
bool zOption = false;
```

```
char xName[100] = { '\0' };
char yName[100] = { '\0' };
char zName[100] = { '\0' };
```

코드에서는 구현의 편의를 위해서 각각의 옵션이 설정되었는지를 나타내는 일종의 플랙(Flag)과 옵션으로 주어진 문자열을 저장하기 위한 부분을 전역 변수를 이용해서 구현했다. 이런 것들은 실제 사용시에는 다른 방법으로 구현되어야 할 것이다. 옵션의 사용이 잘못되었을 경우에는, 제대로 된 사용을 보여주는 것을 항상 만들어 두어야 한다("print_usage()");

```
void parse_arguments(int argc, const char *argv[]) {
    while ((argc > 1) && (argv[1][0] == '-')) {
        switch (argv[1][1]) {
            case 'x':
                printf("%s\n", &argv[1][2]);
                strncpy(xName, &argv[1][2], strlen(&argv[1][2]));
                xOption = true;
                break;
            case 'y':
                printf("%s\n", &argv[1][2]);
                strncpy(yName, &argv[1][2], strlen(&argv[1][2]));
                yOption = true;
                break;
            case 'z':
                printf("%s\n", &argv[1][2]);
                strncpy(zName, &argv[1][2], strlen(&argv[1][2]));
                zOption = true;
                break;
            default:
                printf("Wrong Argument: %s\n", argv[1]);
                print_usage();
                break;
        }
        ++argv;
        --argc;
    }
}
```

커맨드 라인의 해석은 넘겨받는 커맨드 라인의 문자열의 개수가 1개 이상일 경우에만 해당한다. 따라서, 단순히 아무런 옵션을 주지 않고 실행하면 커맨드 라인 해석은 일어나지 않아야 한다. 실행하는 프로그램을 제외하고 첫 번째 옵션은 "argv[1][0]"에서 시작한다. 공백은 문자열에서 제외 되기에, 모든 주어진 문자열은 앞이나 중간에 공백이 없다고 가정할 수 있다. 첫 번째 문자열이 "-"인 경우에만 어떤 옵션인지 검사한다. 만약, 해당하는 옵션 문자가 있다면, 바로 다음에 시작하는 문자열을 화면에 표시하고 ("argv[1][2]"), 옵션이 설정 되었는지를 표시한 후 설정된 문자열을 화면에 표시하기 위해서 저장하도록 한다. 옵션 설정이 잘못 되었다면 사용법을 사용자에게 보여줄 것이다("print_usage()").

그 다음 문자열을 처리하기 위해서는, 문자열의 포인터를 이동시켜준다("++argv"), 그리고, 처리할 문자열의 개수를 줄이기 위해서 카운트 값도 하나 줄인다("--argc"). 증가된 문자열 배열에 대한 포인터는 다시 그 다음 옵션이 있는지를 확인할 수 있는 방법을 제공해 준다("argv[1][0]"). 나머지 커맨드 라인에 대한 해석은 반복된 과정의 연속이다.

```
void print_options(void) {
```

```

if ((xOption) || (yOption) || (zOption)) {
    printf("The Option and Name : ");
    if (xOption) {
        printf("%s ", xName);
    }
    if (yOption) {
        printf("%s ", yName);
    }
    if (zOption) {
        printf("%s ", zName);
    }
    printf("\n");
}
}

int main(int argc, const char *argv[]) {
    puts("Command Line Parsing Example 01");
    printf("Program name: %s\n", argv[0]);
    parse_arguments(argc, argv);
    print_options();

    return EXIT_SUCCESS;
}

```

"main()"함수에는 간단히 해석 함수와 해석된 결과를 출력하는 함수들로 구성되었다. 프로그램을 커맨드 라인에서 실행한 결과는 아래와 같다. 그냥 Eclipse와 같은 곳에서 실행하면, 옵션을 명시할 수 없기에 옵션 해석 내용을 출력하지는 않는다.

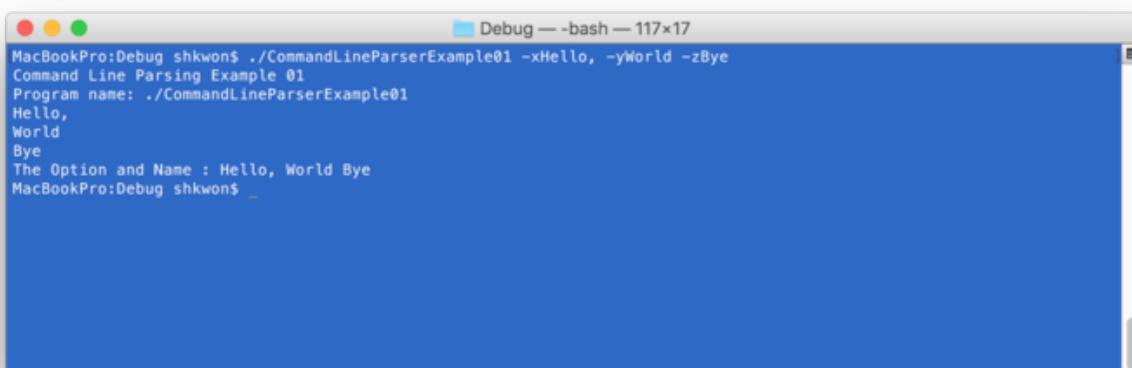
[결과: Eclipse에서 실행]

```

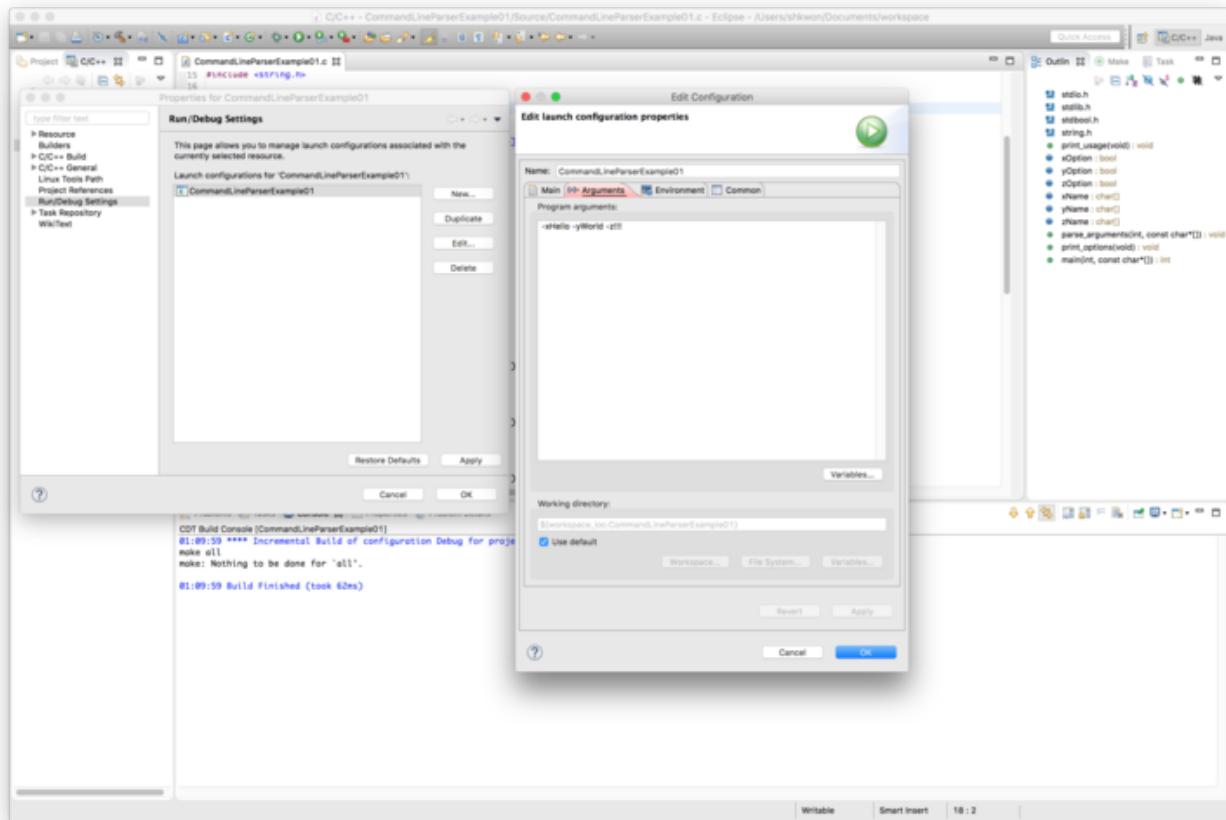
Command Line Parsing Example 01
Program name: /Users/shkwon/Documents/workspace/CommandLineParserExample01/
Debug/CommandLineParserExample01

```

[결과: 커맨드 라인에서 실행]



[참고] 만약, Eclipse에서 커맨드 라인과 같이 실행하고 싶다면, "Project->Properties->Run/Debug Settings->"프로젝트 이름"->Edit->Arguments Tab"에 필요한 옵션을 추가하면 된다.



[직접 하려고 하지 말고 시켜라(역할과 책임의 분리)]

여러 사람이 코딩할 때, 자료구조를 알게되면 그 자료구조를 직접 접근해서 수정하거나, 그것을 이용해서 특정 연산을 하려고 생각하는 경우가 많다. 즉, 자료구조가 정의된 곳을 벗어나, 해당 자료구조를 이용한 연산을 만들려는 시도를 하게된다. 이렇게 만들어진 코드들은 대부분 서로 밀접한 관련성 (Coupling)을 맺는 경우가 많으며, 이는 의도하지 않은 변경에 대해서 약한 구조(오류를 유발하는 구조)를 만든다. 코드간의 의존성으로 인해서 다른 곳에서 발생한 변경의 영향이 자신의 코드까지 도달하게 된다.

이를 방지하기 위해서는 자료구조와 함수를 같이 두는 것이 바람직하며, 될 수 있으면 자료구조를 직접적으로 접근해서 연산하는 것을 막는 것이 좋다. 즉, 자료구조 자체를 상대에게 보여주지 않고, 자료구조에 대한 필요한 연산을 API형식으로 외부에 제공하는 것이다. 물론, API자체의 이름도 자료구조의 내부를 판단할 수 있을 만큼 많은 정보를 주는 것은 좋지않다. 예를 들어, 특정 필드를 직접 설정하고 그 값을 사용하는 등의 연산은 바람직하지 않다.

```
int func( MYTYPE myData ) {
    ...
    return ( getXValue( myData ) + getYValue( myData ) ) * getZValue( myData );
}
```

만약, 위와 같은 코드가 자료구조가 정의된 곳에 있지 않고 외부에 있다고 가정한다면, 이미 자료구조 자체에 어떤 필드들이 있으며, 어떤 역할을 하는지를 외부에 알려져 있다는 뜻이다. 즉, 이것은 정보를 지나치게 많이 외부에 노출한다고 볼 수 있다. 따라서, 정보를 감추기 위해서는 함수("func()")를 자료구조가 정의된 위치한 곳으로 이동시켜주는 것이 더 올바른 선택일 것이다.

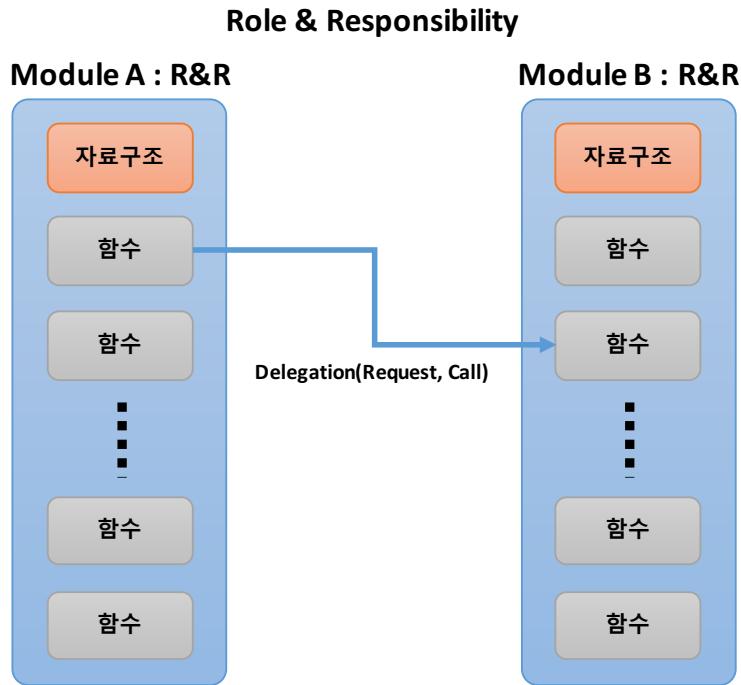
```
int getOperationResult( MYTYPE myData ) {
```

```

    return func( myData );
}

```

"func()" 자체를 자료구조가 선언된 곳으로 이동하고, "func()"을 위부에 공개하는 인터페이스로 정의했다면, 위와 같이 사용자 측의 코드가 작성될 것이다. 물론, 함수 호출이 한번 더 발생하는 것은 사실이지만, 이제는 자료구조가 어떻게 만들어졌는지에 대한 가정 없이 단순히 인터페이스만 호출할 뿐이다. 나중에 인터페이스의 구현 변경이나, 내부 자료구조의 변경에 대해서도 사용자 측의 코드 영향도는 최소화가 될 것이다.



여기서 알 수 있듯이, 직접 자료를 처리하려고 하기보다는, 원하는 일을 상대에게 시키는 방식으로 구현하는 것이 코드의 독립성을 더 높일 수 있다(의존성을 낮춘다). 상대에게 일을 시킨다는 의미는 자신이 다른곳에 할 정보를 알지 못하면, 그 정보를 알고 있는 함수나 모듈에 일을 맡기라는 뜻이다. 그렇게 만들지 않고 직접 자신이 처리하려고 나서면, 자신이 해야 할 역할(Role)이 모호하게 되며 코드의 크기는 추가된 역할 만큼 늘어날 것이다.

또한, 역할의 중복이 발생하게 되어, 필요한 자료구조나 함수를 접근하기 위해서 순환적인 의존관계를 만들 가능성도 높아진다. 일은 그 일을 처리할 역할을 맡은 모듈이나 함수에 맡기고, 우리는 그냥 그것을 이용하는 방향으로 구현을 분리시켜 나가야 한다.

5. 단위 테스트

C언어에서의 단위 테스트란 함수 수준에서 정확히 원하는 목적으로 함수가 만들어졌는지 확인하는 테스트를 말한다. 즉, 코딩 된 함수가 제대로 실행 되는지를 확인하는 것이다. 대부분의 개발자는 단위 테스트가 필요하다는데는 동의하지만, 실제로 단위 테스트를 실행하는 개발자는 드물다(물론, 외국의 사례는 예외). 현재의 개발 방식은 과거와 많은 부분이 달라졌으며, 코딩이 일어나는 시점과 테스트 시점 사이의 간격을 가능한 짧게 가져가는 것이 추세다.

즉, 오류가 발생하는 시점에 수정을 한다는 것이 원칙이다. 이유는 오류가 발생하는 시점과 그것을 발견하는 시점의 차이가 벌어 질수록 수정하는데 들어가는 비용(주로 개발자가 낭비하는 시간 비용 및 제때 출시하지 못해서 발생하는 시장 상실에서 발생하는 이익 손실)이 크게 증가하기 때문이다. 최악의 경우 배포 이후에 발견되는 버그는 요구사항 분석 단계에서 발견하는 버그의 10이상의 비용을 발생 시키기도 한다.



단위 테스트는 마치 제품의 부품을 실험하는 것과 같다고 생각할 수 있다. 부품의 품질이 높으면 만들어진 제품의 품질도 높아질 것은 충분히 예상할 수 있다. 조립이 완료된 제품에서 문제가 생기는 경우, 전체 제품을 다시 분해해서 문제점을 찾아내는 것은 어려운 일이다. 따라서, 조립 전에 부품의 품질을 확보하는 것은 재작업을 줄여 과제 완료시점을 단축하는데도 핵심적인 방법이다. 더군다나, 단위 테스트는 자동화되어 언제라도 실행할 수 있기 때문에, 테스트 비용이 거의 “0”에 가깝다고 생각할 수 있다. 물론, 단위 테스트를 만든데는 비용(테스트를 만들기 위한 시간)이 투자되지만, 반복된 자동화된 테스트는 그 비용을 충분히 만회 하고도 남을 것이 분명하다.

단위 테스트에서 말하는 단위란 무엇일까? C언어에서는 이미 함수라고 이야기 했지만, 문제는 함수의 크기가 다양하다는 점이다. 따라서, 함수를 단위 테스트 한다고 이야기 하기는 쉽지만, 모든 함수가 대상이 될 필요가 없다는 말도 된다. 이를 위해서 어떤 함수를 단위 테스트하는 것이 중요한지 이해해야 한다. 테스트 대상이 될 수 있는 함수는 함수가 정의된 파일 밖에서 접근할 수 있도록 만들어진 것이다. 즉, 파일의 내부에서 사용 되는 함수는 외부에 제공되는 함수를 테스트 하는 과정에서 실행될 기회를 얻게 되며, 리팩토링(Refactoring)과 같은 과정에서 만들어진 많은 인라인(Inline) 함수 들도 독립적으로 테스트 될 필요는 없다.

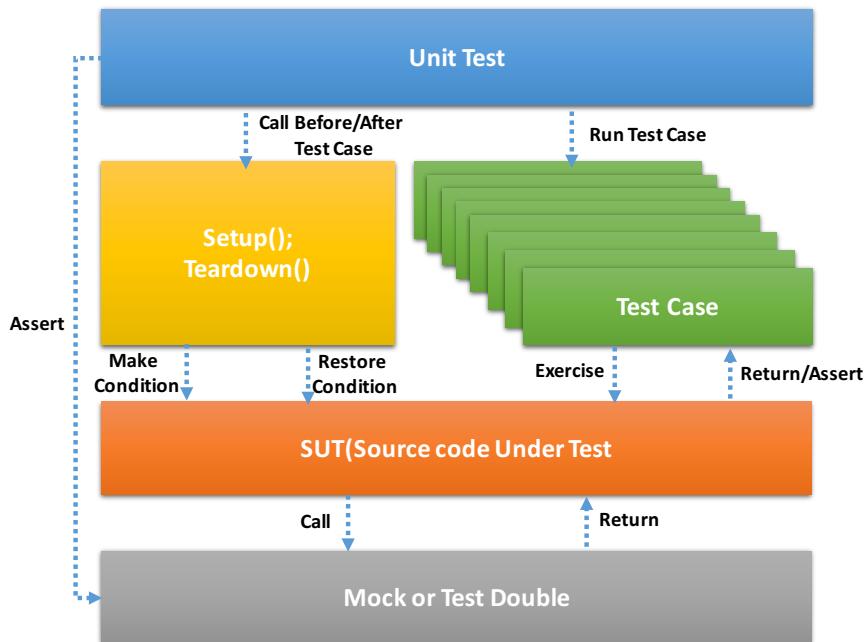
물론, 내부에서 사용하는 함수라고 하더라도 중요한 알고리즘이나 핵심적인 처리를 맡고 있는 경우에는 별도로 단위 테스트 하는 것이 커버리지(Coverage)와 안정성 측면에서는 도움이 될 것이다. 완벽한 테스트가 불가능 하기에, 외부로 제공되는 함수를 통해서 테스트 될 수 있는 코드의 범위도 낮을 가능성이 있기 때문이다. 따라서, 요약하면 외부로 제공되는 함수와 핵심 알고리즘을 구현한 함수는 단위 테스트의 대상으로 생각해야 할 것이다. 참고로 단위 테스트는 95%이상의 문장(Statement) 커버리지를 만족시키는 것이 좋다.

[단위 테스트 프레임 워크]

C언어에서는 기본적인 단위 테스트 단위는 함수다. 즉, 하나의 함수를 다른 함수가 호출해서 그 함수가 올바르게 동작 하는지 검증하는 것을 단위 테스트라고 말한다. 호출하는 함수에서 호출되는 함수의 입력

값과 출력값을 예상하게 되며, 만약 예측된 값이 제대로 출력되지 않는다면, 단위 테스트는 실패라고 본다.

테스트의 대상이 되는 코드를 SUT(Software Under Test, 혹은 Source Code Under Test : Production Code)라고 하며, 호출하고 검사하는 측을 테스트 케이스(Test Case)라고 한다. 또한, 테스트 되는 대상의 코드가 의존하고 있는 다른 함수들도 있는데, 단위 테스트에서는 그런 함수들을 테스트 더블(Test Double)을 사용해서 대체한다. 즉, 테스트 되는 함수는 제한된 상황에서 실행된다고 보면 된다. 테스트를 하기 위해서 부대 상황을 만들어주는 코드를 픽스처(Fixture)라고 부르며, 테스트 되는 함수가 호출하는 함수들을 대체하기 위해서 목(Mock)을 사용한다. 따라서, 테스트를 마치 실제 코드의 실행 시나리오대로 동작 하는지 검증할 수 있다. 단위 테스트를 자동화해서 실행시킬 수 있는 환경을 제공하는 도구를 테스트 프레임워크라고 부르며, 단위 테스트를 위한 각종 편의를 제공한다.



단위 테스트를 실행하기 위해서는 단위 테스트를 동일한 특정 기능에 한정된 묶음으로 테스트 케이스들을 모을 수 있는데, 이를 테스트 그룹(Test Group)이라고 한다. 하나의 테스트 그룹에는 여러 개의 단위 테스트 케이스들이 들어가며, 각각은 다른 이름으로 표현된다. 테스트 케이스의 이름은 테스트하는 내용이 무엇인지 정확히 기술하기 위해서 주어지는 이름표일 뿐이다.

하나의 단위 테스트 그룹에는 설정(Setup)과 복구(Tear-down)가 각각의 테스트 케이스에 대해 반복적으로 실행된다. 따라서, 테스트 케이스의 실행에 필요한 선형 조건(Pre-Condition)을 생성하고, 이후에 테스트 케이스 간의 의존성을 없애기 위해서 복구 절차를 만들 수 있다. 대체 함수 역할을 하는 목(Mock)은 호출해야 할 함수의 이름과 그것의 파라미터 타입 및 리턴 타입, 파라미터에 전달되어야 값과 리턴 값을 기록(저장)한다.

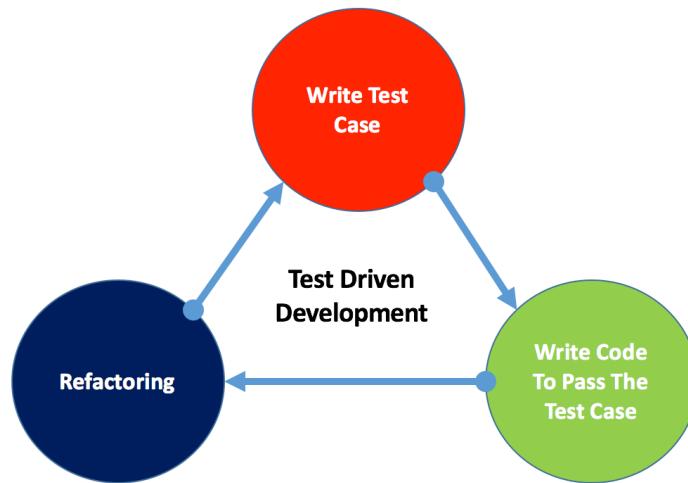
대체 함수의 호출이 발생하면, 기록(저장)된 기대값을 불러 와서, 테스트 중인 호출 함수로 돌려주게 된다. 따라서, 각각의 테스트 케이스에서는 상황에 맞게 목(Mock)의 기대값을 설정하고, 이후에 원하는 기대값들이 제대로 나왔는지 확인하는 과정을 거친다. 만약, 기대한 설정한 값들이 사용이 되지 않았거나, 테스트 중인 함수가 정해놓은 파라미터 값으로 호출하지 않았다면, 테스트 케이스는 실패한 것으로 간주 한다. 실패한 테스트 케이스라고 단위 테스트가 즉각 끝나지는 않는다. 이러한 결과들은 모아서 나중에 출력으로 보여주게 된다.

목(Mock)은 단위 테스트의 실행을 관리하는 테스트 프레임워크와는 별도로 존재할 수 있으며, 하나의 단위 테스트 프레임워크에서 선호하는 목(Mock)도 같이 배포할 수도 있다. 따라서, 자신이 익숙한 목(Mock)이 있다면, 그것을 사용하거나 더 좋은 목(Mock)을 선택해서 사용해도 된다. 하지만, 언어적인 제약이 있는 경우에는 물론 사용하지 못할 것이다.

보통의 경우 단위 테스트 프레임워크는 "xUnit"과 같은 형식의 이름을 가진다. 여기서 "x"는 사용하는 프로그래밍 언어를 나타낸다. 단위 테스트는 Java에서 먼저 시작되었지만, 그 유용함으로 인해서 다양한 언어에서 구현 되었으며, 애자일(Agile)과 같은 방법론에서는 코드의 검증을 빨리 하기 위해서 단위 테스트를 필수적인 활동으로 포함하고 있다. 따라서, 현대적인 소프트웨어 개발자라면 단위 테스트를 반드시 익혀야 할 부분이라고 생각해야 할 것이다.

단위 테스트와 같이 사용할 수 있는 테크닉으로는 디자인 패턴과 리팩토링(Refactoring)이 있으며, 이들을 묶어서 테스트를 먼저 정의하는 방식으로 코딩을 이끌어 가는 방식을 TDD(Test Driven Development)라고 부른다. TDD가 좋은 점은 소프트웨어의 디자인을 개선하는 효과가 있다는 점과 필요한 코드만 구현하도록 만들어 준다는 점이다. 즉, 테스트를 통과하기 위해서 필요한 코드를 구현하기에 명세에 딱 맞을 만큼의 코드만 구현하게 되며, 구현된 코드가 테스트를 하기 편하게 구현되어 간략하게 만들어진다는 점이다.

따라서, 개발자는 기존의 방법에서 부족했던 이런 부분들을 TDD를 통해서 익혀나갈 수 있으며, 과제 측면에서는 검증된 코드를 안정적으로 확보할 수 있다는 점에서 도움을 얻을 수 있다. 물론, 자동화를 통해서 만들어진 테스트의 생산성은 시간이 지날수록 더 높아진다. 추가적으로 변경에 대한 두려움도 줄여줄 수 있다는 점에서 단위 테스트를 핵심적인 개발 활동으로 포함하는 것이 좋을 것이다.



단위 테스트가 실무에 정착하지 못한 이유는 개발자의 책임이다. 개발자가 스스로의 업무를 적극적으로 정의하지 않았기 때문에 발생한다고 본다. 즉, 코드의 품질을 확보하는 과정을 업무로서 정의하지 않고, 테스트를 전담하는 사람들에게 맡기는 경향에서 찾을 수 있다. 기본적으로 자신이 작성한 코드가 제대로 동작하는 것을 보장하는 것은 코드를 만든 사람의 책임이다. 완벽히 동작하는 것을 보장 하라는 것이 아니라, 통합 되어도 크게 문제가 없을 수준으로 동작하도록 만들라는 것이다.

하지만, 현재는 대부분의 코드가 통합 이후에 테스트를 한다. 이때는 복잡한 코드에서 발생하는 문제점을 기록하고, 재현하고, 그것의 원인을 밝힌 후, 다시 수정하는 과정을 거치게 된다. 이때 재작업(Rework)으로 인한 추가적인 버그가 발생할 가능성(Side Effect)도 열어두고 있는데, 단위 테스트는 이를 방지할 수 있는 "안전망(Safety Net)" 역할을 할 수 있다. 단위 테스트가 활성화 되기 위해서는 과제 책임자의 적극적인 의지와 지원이 필요하며, 개발자들은 스스로 단위 테스트 활용 능력을 키워야 한

다. 디자인 패턴, 리팩토링, TDD, Coding Rule 등등 다양한 부분들의 역할을 꾸준히 함께 성장시켜 나가야 할 것이다.

단위 테스트는 개발자의 개발 활동이지만, 그것을 지원하지는 관리자가 없다면 제대로 개발에 반영되지 않을 것이 분명하다. 관리자가 과제의 상황을 파악하기 위해서 개발자에게 몇 %나 일이 진행되었는지 묻는 것은 무의미한 질문이다. 오히려 과제의 정확한 현재 상황을 알기 위해서 필요한 질문은 “각각의 요구사항에 대한 테스트 케이스의 존재 여부와 실행 결과”를 보는 것이 현명하다.

[하드웨어가 없으면 코딩할 수 없다?]

보통 하드웨어가 준비되지 않으면, 실제 하드웨어에서 동작해야하는 소프트웨어를 만든데 어려움을 겪는 경우가 많다. 대부분의 경우 사용하려는 칩셋(Chipset)으로 만들어진 평가보드(Evaluation Board)를 구매해서 일단 소프트웨어 개발을 진행하게 된다. 요즘 같은 경우에는 대부분의 하드웨어 기능이 하나의 칩에 다 들어가 있는 경우가 많지만, 특정 주변장치를 사용하기 위한 인터페이스 역할을 하는 칩들은 따로 구매해서 사용하기도 한다.

예를 들어, 네트워크 디바이스같은 것은 하나의 주요 칩에 들어가지 않고 따로 구매해서 사용한다. 그리고, 특정 장치의 인터페이스를 위해서 메모리 주소 맵핑(Memory Mapped I/O)을 하는 경우도 있는데, 이럴 경우에는 하드웨어와 직접적인 통신을 위해서 메모리 공간에 대한 접근을 시도하게 된다. 즉, 하드웨어와 직접적인 통신을 담당하는 소프트웨어의 경우에는 타겟(Target)이 되는 하드웨어가 없는 상황에서는 소프트웨어의 개발이 어렵다.

그렇다고, 그냥 하드웨어의 개발이 끝나기를 기다리는 것은 과제를 자연시킬 가능성이 많기에, 대부분의 경우 하드웨어 인터페이스를 계층적으로 분리하는 방법으로 소프트웨어 개발을 시도하게 된다. 즉, 하드웨어의 스펙(Spec.)이 있다면, 이것을 기준으로 하드웨어에서 제공 해야하는 기능과 소프트웨어에서 제공 해야할 데이터를 분석한다. 하드웨어 스펙을 구할 수 없다면, 해당하는 하드웨어와 유사한 기능을 하는 하드웨어를 찾아서 제공되는 기능과 제공해야할 데이터를 보고, 하드웨어에 접근하는 API(Application Programming Interface)를 먼저 정의하게 된다(이것을 HAL: Hardware Abstraction Layer라고 부르기도 한다.).

원칙은 하드웨어를 추상화 시키는데 있다. 즉, 하드웨어에 대한 직접적인 접근 대신에 그에 해당하는 인터페이스를 정의하고, 이를 기반으로 소프트웨어를 만드는 것이다. 따라서, 실제 하드웨어가 나오게 되면 정의한 인터페이스를 구현하는 일(포팅:Porting)을 해주어야 한다. 하드웨어가 준비되기 전에는 마치 실제 하드웨어와 같이 동작하는 가상의 환경을 소프트웨어로 구현해서 사용한다. 여기서 중요한 점은 직접적인 하드웨어가 맵핑(Mapping)된 주소를 접근하는 의존적인 코드를 만들어서는 안된다는 것이다.

하드웨어에 정확한 값을 읽고 쓰는지 확인하기 위해서는, 하드웨어 접근을 위해서 필요한 주소를 외부에서 지정하는 방식을 사용하는 것이다. 즉, 하드웨어 인터페이스를 가로채서 원하는 값을 읽고 쓰는지 확인하기 위해서다. 이것은 일종의 "의존성 삽입(Dependency Injection)" 기법으로, 객체지향 언어에서 객체의 생성을 특정 객체 내부에서 하기보다 외부에서 삽입해 주는 방법을 응용한 것이다. 당연히 삽입되는 객체나 혹은 하드웨어에 의존적인 주소 값들은 외부에서 정의한 객체나 변수의 주소 등이 될 것이다. 따라서, 외부에서는 해당 객체나 변수를 읽어서 원하는 동작을 하는지 확인할 수 있으며, 특정한 경우에는 인위적으로 원하는 값을 적어줄 수도 있다.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

unsigned hardwareStatusRegister;
```

```

bool initializeHardware( unsigned &statusRegisterAddress ) {
    *statusRegisterAddress = 0xFFFF0000; /* Initialize Hardware with 0xFFFF0000 */
    if ( *statusRegisterAddress != 0xFFFF0000 ) {
        return FALSE;
    }
    return TRUE;
}

int main() {
    if ( initializeHardware(&hardwareStatusRegister) == FALSE ) {
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}

```

위의 코드는 특정 주소에 값을 저장하고, 그것을 읽어서 쓴 값과 읽은 값이 일치 하는지 확인해서 제대로 하드웨어가 초기화 되었는지 확인하고 있다. 실제로는 하드웨어의 레지스터들은 읽기 전용이나 쓰기 전용, 혹은 읽기 쓰기 가능 등으로 나눠질 수 있으며, 읽기나 쓰기가 불가능한 경우도 있다. 여기서는 하드웨어의 상태 레지스터의 주소를 초기화 시에 삽입(Inject)시켜서, 나중에 다른 곳에서 해당 변수를 읽을 수 있도록 만든 경우다.

[테스트를 위한 코드 작성 법]

여러가지 소프트웨어 품질을 측정하는 기준이 있지만, 그중에서도 가장 기본이 되는 것은 테스트를 할 수 있는가를 따지는 것이다. 일명 "Testability"라고도 하며, 얼마나 쉽게 테스트가 가능한지를 본다. 물론 여기서 보는 것은 개인의 주관적인 부분이 아니라, 코드가 얼마나 테스트가 되고 있는가를 객관적인 기준으로 살핀다. 어떤 언어를 사용하든 상관없이 "단위 테스트"는 개발자가 익혀야 할 기본이다. 따라서, 단위 테스트가 쉬운 코드를 어떻게 만들 수 있는지 알아야 할 것이다.

C언어에서의 단위 테스트는 하나의 함수를 대상으로 한다. 따라서, 하나의 함수당 최소한 하나의 단위 테스트가 있다고 생각할 수 있다. 물론, 모든 함수마다 단위 테스트가 필요한 것은 아니다. 하지만, 일단은 하나의 함수를 만들면 하나의 단위 테스트를 만든다고 생각하면 된다(반대로 테스트를 먼저 만들고, 테스트를 통과하기 위해서 코드를 짜는 것이 TDD다).

함수에 대한 입력의 개수가 늘어나면, 당연히 테스트 케이스도 늘어나야 한다. 입력의 값이 변화가 크다면, 테스트로 사용될 데이터의 양도 커질 것이다. 모든 입력의 범위를 다 테스트하는 것은 불필요 하기에 "경계 범위(Boundary Condition)을 적절히 입력 값으로 사용할 수도 있을 것이다. 혹은, 범위를 가지고 있는 값의 경우에는 각각의 범위에 해당하는 대표적인 값을 사용할 수도 있다.

예를 들어, 0에서 100까지의 값을 가지는 입력을 함수가 받는다면, "-1, 0, 1, 99, 100, 101"과 같은 경계를 이루는 값과 경계내의 수치 값들을 테스트의 입력으로 사용할 수 있다. 만약, 이런 입력 변수들의 늘어난다면, 당연히 테스트 해야할 조합의 수도 늘어나게 된다. 따라서, 단위 테스트가 쉬운 함수란 입력 파라미터의 수가 적을수록 좋다. 입력 파라미터가 하나도 없다면 가장 좋다. 물론, 이때는 전역변수나 파일범위 변수를 사용할 가능성도 있기에, 다른 함수와 완전한 분리가 안될 가능성도 높다(일반적으로 전역 변수는 좋은 선택이 아님). 어쨌든 입력이 없으므로 초기 조건 설정에 따른 출력값 만을 확인하면, 테스트는 종료될 가능성이 높다(물론, 사용하는 변수의 범위에 따라, 초기 조건의 설정이 어려울 수도 있음).

초기 조건의 설정이 함수의 내부에서 이루어진다면, 제대로 설정이 되었는지 외부에서 알기 어려울 수 있다. 또한, 하드웨어에 대한 출력이 함수 내부에서 이루어진다면, 그 함수가 제대로 외부로 출력을 내

보내는지도 알기 어렵다. 이를 위해서는 함수가 내부적으로 사용하는 것들에 대해서 감시를 할 수 있어야 하며, 어떤 경우에는 외부에서 특정 변수의(혹은, 메모리의) 주소 값을 주어서, 그 변수에 올바른 값이 적히는지를 확인할 필요도 있다.

함수의 길이가 길면 테스트 케이스도 늘어날 가능성이 높다. 또한, 조건 분기문이 늘어날수록 테스트 케이스도 늘어난다. 따라서, 함수는 짧아야 하고, "if()"나 "do~while()", "switch()"와 같은 것이 적을수록 테스트하기 쉬운 함수가 된다. 분기문이 많으면 많을수록 경우의 수를 조합해야 하고, 각각의 경로가 수행되기 위한 필요한 데이터를 외부에서 제공해 주어야 한다. 또한, 하나의 함수가 다른 함수와 연결이 많은 경우, 해당 함수만 따로 떼어서 테스트 하기가 어려워 지며, 이때는 함수 내부에서 호출되는 함수 들에 대해서 목(Mock)을 제공해 주어야 한다. 따라서, 테스트 하기 좋은 함수는 외부의 함수 호출이 적을수록(의존성이 낮을 수록) 좋을 것이다. 즉, 외부 코드에 대한 의존성이 높을수록 테스트는 어렵게 된다.

만약, 이미 작성된 코드를 단위 테스트를 하고자 한다면, "Leaf Function"에 해당하는 함수들 부터 시작하는 것이 좋다. 즉, 그런 함수들은 외부 코드와 의존성이 없으며, 호출 경로상 가장 말단에 위치한 함수들이다. 즉, 가장 하위에 속한 것들을 먼저 골라서 단위 테스트를 시작하는 것이 좋을 것이다. 테스트 범위를 확인하는 방법은 "GCOV"와 같은 커버리지(Coverage)측정 툴을 이용할 수 있다. 실행되지 않은 코드가 있다면 테스트 케이스를 보완 해야하고, 테스트가 성공했다는 것을 보장하도록 코드를 유지해야 한다. 물론, 모든 코드가 실행 되었고 성공적인 결과를 낸다고 해서, 향후에 문제가 생길 가능성이 전혀 없다는 것은 아니다. 단위 테스트는 해당 함수가 오류가 없다는 것을 보여줄 뿐이지, 다른 함수의 결합에서 발생하는 문제까지 전부 제거해 주는 것은 아니다.

테스트는 빨리 실행되어야 한다. 특히, 단위 테스트는 빨리 결과를 받아볼 수 있어야 한다. 하나의 단위 테스트가 시간이 1분이 걸린다면, 100개로 늘어나게 되면, 1시간 40분이나 된다. 이런 식이라면 단위 테스트 전체를 실행하는데, 몇 일이 걸리거나 혹은 몇 주가 걸릴지도 모른다. 따라서, 하나의 함수가 오래 동안 수행되어야 테스트가 가능한 경우(특정 시간, 특정 선행조건, 하드웨어 오류 검사 등)는 인위적인 환경에서 검증할 수 있는 방법을 만들어야 한다. 즉, 함수 자체가 외부 환경(타이머, 네트워크, 파일등의 자원)에 노출될 때, 직접적으로 어떤 것들을 다루기 보다는 API와 같은 통해서 연결되도록 만들어야 할 것이다. 그렇지 않고 직접 외부로 접근한다면, 가상 환경을 만드는데 어려움을 겪을 수 있기 때문이다. 특히, 하드웨어와 관련된 오류 등을 제대로 처리 하는지를 검증하는 것은 이런 가상 환경을 기반으로 코딩하는 것이 좋다.

[의존성의 주입(Dependency Injection)]

의존성 주입이란 외부에서 내부의 코드가 의존하고 있는 것을 호출을 통해서 주입(Injection)해 주는 것을 말한다. 예를 들어, 어떤 함수가 특정 자원에 접근해야 하는 상황이라면, 내부 코드에서 직접적으로 접근할 수도 있지만, 이를 외부에서 함수의 호출 시에 알려주는 방법을 적용하는 것이다. 이를 통해서 얻게 되는 이점은 해당 자원에 대한 가상화가 가능해, 실제 자원이 없는 상황에서도 코드를 실행해 볼 수 있다는 것이다.

만약, 코드의 내부에서 특정 자원을 접근한다면 외부에서는 이를 알 수 없다. 따라서, 그런 함수를 테스트 하기 위해서는 필요한 자원을 반드시 만들어 미리 제공해 주어야 하며, 그렇지 않다면 제대로 된 테스트가 불가능하다. 또한, 그 자원에 해당 함수가 제대로 접근하고 있는지도 알지 못하며, 실제 구현이 없을 경우에는 빠르게 테스트를 실행하는 것도 불가능해 질 수 있다. 따라서, 이런 경우에는 외부에서 "의존하고 있는 것(Dependency)"을 직접 접근할 수 있는 방법을 "주입(Inject)"하는 것으로 해결해야 한다.

```
#include <stdio.h>
#include <stdlib.h>
```

```
typedef struct message MESSAGE;
```

```

struct message {
    char *message;
};

typedef struct handler HANDLER;
struct handler {
    void (*handle)(MESSAGE *msg);
};

void handler_A(MESSAGE *msg) {
    printf("This is HANDLER A : %s\n", msg->message);
}

void handler_B(MESSAGE *msg) {
    printf("This is HANDLER B : %s\n", msg->message);
}

typedef struct dependencyinjection DEPENDENCY_INJECTION;
struct dependencyinjection {
    HANDLER *handler;
    void (*init)(DEPENDENCY_INJECTION *di, HANDLER *handler);
    void (*do_handle)(DEPENDENCY_INJECTION *di, MESSAGE *msg);
};

void init(DEPENDENCY_INJECTION *di, HANDLER *handler) {
    if ((di != NULL) && (handler != NULL)) {
        di->handler = handler;
    }
    return;
}

void do_handle(DEPENDENCY_INJECTION *di, MESSAGE *msg) {
    if ((di != NULL) && (di->handler != NULL)) {
        di->handler->handle(msg);
    }
    return;
}

DEPENDENCY_INJECTION di = { NULL, init, do_handle };
HANDLER a = { handler_A };
HANDLER b = { handler_B };

int main(void) {
    puts("Dependency Injection Example 01");
    MESSAGE msg = { "Hello, World!!!" };
    di.init(&di, &a);
    di.do_handle(&di, &msg);
    di.init(&di, &b);
    di.do_handle(&di, &msg);

    return EXIT_SUCCESS;
}

```

[결과]

```
Dependency Injection Example 01
This is HANDLER A : Hello, World!!!
This is HANDLER B : Hello, World!!!
```

위의 코드는 특정 메시지를 처리(handle)하기 위한 핸들러를 인위적으로 외부에서 설정할 수 있도록 만들어준 경우다. 동일한 메시지를 받더라도 현재 설정된 핸들러에 따라 다른 처리를 할 수 있다. 물론, 외부에서 전달해준 핸들러를 이용해서 다양한 처리를 더 추가해 나갈 수도 있다.

[의존성의 주입의 활용]

이번에 보게 되는 예제는 제품의 코드와 같이 특정 모델에서 정의된 행동을 보여야 할 경우에 응용해 볼 수 있을 것이다. 즉, 동일한 이벤트(Event)라고 하더라도 모델이나 상태에 따라 다른 처리를 해야 할 경우에 사용할 수 있는 방법이다. 핸들러를 설정하고, 다른 핸들러로 동적으로 교체하는 것까지 포함하고 있다.

```
#ifndef MESSAGE_H_
#define MESSAGE_H_
typedef struct message {
    char *message;
} MESSAGE;

#endif /* MESSAGE_H_ */

/* Handler.h */
#ifndef HANDLER_H_
#define HANDLER_H_
#include "Message.h"

typedef enum {
    HANDLER_TYPE_A = 0, HANDLER_TYPE_B, HANDLER_NONE,
} HANDLER_TYPE;

typedef struct handler HANDLER;
void handle(HANDLER *handler, MESSAGE *msg);
HANDLER *getHandler(HANDLER_TYPE type);
#endif /* HANDLER_H_ */

/* Handler.c */
#include <stdio.h>
#include "Message.h"
#include "Handler.h"
struct handler {
    void (*handle)(MESSAGE *msg);
};

void handler_A(MESSAGE *msg) {
    printf("This is HANDLER A : %s\n", msg->message);
}

void handler_B(MESSAGE *msg) {
```

```

printf("This is HANDLER B : %s\n", msg->message);
}

HANDLER handler_type_a = { handler_A };
HANDLER handler_type_b = { handler_B };

HANDLER *getHandler(HANDLER_TYPE type) {
    switch (type) {
        case HANDLER_TYPE_A:
            return &handler_type_a;
            break;
        case HANDLER_TYPE_B:
            return &handler_type_b;
            break;
        default:
            return NULL;
            break;
    }
}

void handle(HANDLER *handler, MESSAGE *msg) {
    handler->handle(msg);
}

/* DependencyInjection.h */
#ifndef DEPENDENCYINJECTION_H_
#define DEPENDENCYINJECTION_H_
#include "Message.h"
#include "Handler.h"

typedef struct dependencyinjection DEPENDENCY_INJECTION;

DEPENDENCY_INJECTION* createDependencyInjection(HANDLER *handler);
void run_handler(DEPENDENCY_INJECTION *di, MESSAGE *msg);

#endif /* DEPENDENCYINJECTION_H_ */

/* DependencyInjection.c */
#include <stdio.h>
#include "Message.h"
#include "Handler.h"
#include "DependencyInjection.h"

struct dependencyinjection {
    HANDLER *handler;
};

DEPENDENCY_INJECTION di = { NULL };

DEPENDENCY_INJECTION* createDependencyInjection(HANDLER *handler) {
    if (handler != NULL) {
        di.handler = handler;
}

```

```

    }
    return &di;
}

void run_handler(DEPENDENCY_INJECTION *di, MESSAGE *msg) {
    if ((di != NULL) && (di->handler != NULL)) {
        handle(di->handler, msg);
    }
}

/* Main.c */
#include <stdio.h>
#include <stdlib.h>
#include "Message.h"
#include "Handler.h"
#include "DependencyInjection.h"

int main(void) {
    puts("Dependency Injection Example 01");
    MESSAGE msg = { "Hello, World!!!" };

    HANDLER *handler = getHandler(HANDLER_TYPE_A);
    DEPENDENCY_INJECTION *di = createDependencyInjection(handler);
    run_handler(di, &msg);

    handler = getHandler(HANDLER_TYPE_B);
    di = createDependencyInjection(handler);
    run_handler(di, &msg);

    return EXIT_SUCCESS;
}

```

[결과]

Dependency Injection Example 01
This is HANDLER A : Hello, World!!!
This is HANDLER B : Hello, World!!!

이와 같이 동적으로 특정 모델이나 기능에 맞게 핸들러를 변경해 주는 것은 흔히 실무에서 만날 수 있는 상황이며, 일종의 생성과 관련된 패턴으로도 볼 수 있다. 즉, 특정 제품을 만들 때 기존의 동작을 변경해 주어야 할 경우 이런 식으로 대응할 수 있다. 좋은 점은 특정 모델에 맞게 분기하는 조건문을 한 곳에 모아서 관리할 수 있다는 점이 있지만, 함수 포인터와 같은 것을 많이 사용한다는 점에서 가독성이 낮아질 가능성도 있다.

[테스트를 위한 코드 작성 규칙]

어떤 함수는 테스트하기 쉽고, 어떤 함수는 테스트 하기 어려운 경우가 있다. 따라서, 단위 테스트를 할 수 있도록 만들려면, 세심하게 고려해서 함수를 구현해 주어야 한다. 그렇지 않다면 테스트 자체가 불가능하거나, 혹은 테스트를 하기 위한 전제 조건을 설정 하느라 많은 노력이 낭비될 수도 있다. 따라서, 테스트를 쉽게 하기 위한 함수를 만드는 방법을 충분히 잘 익혀서 활용해야 할 것이다.

01. 함수의 리턴 값을 만들라.

; 리턴 값이 없는 함수는 제대로 동작하는 것을 확인하기 위해서는 내부의 상태를 뽑아내야 한다.

02. 함수의 파라미터의 수를 최소한으로 만든다.

; 파라미터의 수가 많아지면 테스트 케이스도 많아진다. 될 수 있으면 파라미터를 없도록 만드는 것이 원칙적으로 좋다.

03. 함수는 짧고 한 가지 명확한 일만 하도록 만든다.

; 여러가지 일을 하는 함수는 길게되며 복잡하게 될 가능성이 높다.

04. 함수 내부의 분기문은 최소화 시킨다.

; 내부 분기문이 많아지면 테스트 케이스도 늘어난다.

05. 함수 내부에서 생성되는 것을 가능한 줄인다.

; 함수 내부에서 생성되는 것들이 있으면, 각각을 다 확인할 수 있는 방법이 필요하다.

06. 함수가 다른 함수를 호출하는 것을 가능한 줄인다.

; 의존성이 늘어나기에 함수 자체를 테스트하는 것에 노력이 더 많이 들어간다.

07. 함수가 사용하는 전역변수를 없앤다.(혹은, 최소한으로 유지한다.)

; 전역 변수가 있으면 외부에서 선행 조건을 만들어 주어야 한다.

08. 함수에 중복된 코드를 없앤다.(혹은, 새로운 함수로 중복을 대체한다.)

; 중복된 코드는 추가적인 함수로 분리해서 따로 검증한다.

09. 함수를 될 수 있으면 전역 함수로 만들지 않는다.

; 내부에서만 사용되는 함수는 내부 함수로 정의해야 한다.

10. 함수가 특정 상태에 민감하지 않도록 만든다.

; 상태에 의존적인 함수는 선행 조건을 미리 만들어 주어야 한다. 또한, 병렬적으로 실행되는 경우에는 문제를 일으킬 가능성도 높다.

이상과 같이 10가지 방법은 다소 인위적으로 만든 것 이기에 필요하다면 더 추가할 수 있다. 이제 간단한 예를 통해서 각각에 대해서 알아보도록 하자.

01. 함수의 리턴 값을 만들라.

함수가 리턴 값이 없으면 어떻게 실행 되었는지 알 수 없다. 물론, 리턴 값이 없는 함수도 존재한다. 그리고, 정말 아무런 리턴 값이 필요없는 경우도 있다. 하지만, 이때는 내부적으로 상태의 변경을 만들었거나, 혹은 외부로 출력을 내보낸 경우다. 따라서, 이런 함수들을 테스트 하기 위해서는 외부에서 검사할 수 있는 추가적인 일이 필요하다.

```
void function( int x ) {
    ...
    return;
}
```

위와 같은 함수가 있다면, 이 함수가 제대로 실행 되었는지 확인하는 것은 어렵다. 즉, 내부의 특정 변수를 외부에서 관찰해야 한다는 말이다. 차라리 아래와 같이 만들면, 함수를 뜯어보지 않고도 제대로 실행되었는지 확인할 수 있다.

```
int function( int x ) {
```

```

...
return xxx;
}

ASSERT( y, function( x ) );

```

즉, 함수의 호출 결과를 통해서 ASSERT()와 같은 것을 이용해서 입력에 대한 결과값 확인을 할 수 있게된다. 만약, 여러가지 입력이 존재할 경우, 각각에 대해서 정확한 값이 나오는지 확인할 수 있다.

02. 함수의 파라미터의 수를 최소한으로 만든다.

함수의 파라미터는 없는 것이 가장 좋다. 하지만, 현실적으로 함수는 파라미터 들을 가지기에, 최소한의 수를 가지도록 만들어야 할 것이다.

```

int function( void ) {
...
return xxx;
}

```

만약, 함수가 외부의 상황에 의존적이지 않다면, 함수에 대해서 파라미터가 없을 경우에는 테스트 케이스는 하나 밖에 존재하지 않는다. 따라서, 테스트 하기도 쉽다. 함수의 파라미터 수가 늘어날 수록 테스트 케이스는 급격하게 늘어날 가능성이 있다. 따라서, 될 수 있으면 함수의 파라미터 수는 적은게 좋다. 없는 것이 가장 좋다. 물론, 없을 수는 없기에 파라미터를 두어야 하지만, 그래도 가능한 줄여야 한다. 4개보다 많아질 경우에는 함수 호출 오버헤드는 더 커질 것이고, 함수의 사용법도 어려워져서 잘못 사용할 가능성도 높아지게 된다.

03. 함수는 짧고 한 가지 명확한 일만 하도록 만든다.

하나의 함수는 한 가지 일만 해야한다. 여러가지 일을 하게 만들면 그 만큼 파라미터의 수도 늘어나고 코드도 길어지게 된다. 여러가지 일을 하기 위해서 조건문 들이 추가될 것이며, 당연히 테스트 케이스도 그 조건을 실행하기 위해서 늘어나게 된다.

```

int do_something_and_do_something_others( int x ) {
    if ( x == '1' ) {
        ...
    }
    if ( x == '2' ) {
        ...
    }
    ...
    return xxx;
}

```

위의 코드가 잘못되었다는 것이 아니라, 여러가지 일을 처리하도록 만들어서 안된다는 것을 표현할 뿐이다. 일반적으로 함수 내부에서 일어나는 일을 한마디로 표현할 수 있는 "함수의 이름"을 찾을 수 있다면, 한 가지 일만한다고 생각할 수 있다. 만약, 한 가지 이름을 부여하기 힘들거나 너무 일반적인 이름 (Common, Helper, Util, Send_and_Check, ...)을 사용하는 함수라면, 여러가지 일을 하나의 함수에서 하고 있을 것이라고 예상할 수 있다.

04. 함수 내부의 분기문은 최소화 시킨다.

분기문이 많으면 테스트 하기 어렵다. 즉, 테스트 케이스를 많이 만들어야 한다. 컴파일러의 입장에서도 최적화하기 힘들며, 캐시(Cache)의 효율도 떨어진다. 물론, 그것에 대비하는 여러가지 하드웨어적인 지원이 있지만, 어쨌든 분기문을 최소화하는 것이 좋다.

```
int function( int x ) {
    if ( x == xxx ) {
        ...
    }
    else if ( x == jjj ) {
        ...
    }
    ...
    return yyy;
}
```

위와 같이 함수를 만들었다면, 최소한 “x”는 “xxx”와 “xxx”가 아닌 값, 그리고, “xxx”이고 “jjj”인 경우, “xxx”이고 “jjj”가 아닌 경우를 고려한 테스트 케이스가 만들어져야 한다. 또한, “x”가 가지는 경계값에 대해서도 정확히 반응하는지를 살피기 위해서 “xxx-1”, “xxx”, “xxx+1”과 같은 값이 사용될 수도 있다. 추가적인 파라미터가 있고, 그 값도 분기문에 영향을 줄 경우에는 테스트 케이스가 더 복잡해 질 가능성이 높다. 물론, 모든 분기문을 제거하는 것은 어렵다. 하지만, 좀 더 간단히 표현하거나, 분기문을 없애는 다른 방법을 가져다 사용할 수도 있을 것이다(리택토링의 “Factory Method”와 같은 방법 활용).

05. 함수 내부에서 가능한 생성되는 것을 줄인다.

함수 내부에서 생성해서 사용할 경우에는 외부에서 그 값을 관찰하기 어렵다. 예를 들어, 특정 하드웨어의 주소에 값을 써야하는 경우, 정확한 값을 쓰는지를 확인하지 못하게 되며, 또한 하드웨어에 대한 의존성으로 인해서 단위 테스트를 실행하지 못하게 될 수도 있다. 따라서, 이런 경우에는 외부에서 의존성을 삽입하는 방법으로 해결 수 있다(Dependency Injection).

```
int CONTROL_REGISTER = 0;

int function( int* x ) {
    ...
    *x = xxx;
    ...
    return yyy;
}

void main( void ) {
    ...
    ASSERT( zzz, function( &CONTROL_REGISTER ) );
    ASSERT( kkk, CONTROL_REGISTER );
}
```

위와 같이 만들어진 "function()"이 제대로 된 값을 "CONTROL_REGISTER"에 쓰는지를 확인하기 위해서, 외부에서 주소 값을 전달한 경우다. 즉, 외부에서 함수가 의존하는 부분을 삽입(Injection)시켜 주었다. 나중에 함수를 호출한 후 제대로 된 값이 적혀있는지를, 외부에서 삽입된 변수의 값과 원하는 값이 나왔는지를 확인하는 과정을 통해서 함수가 정의된 목적으로 동작한다는 것을 보장할 수 있다.

만약, 내부에서 하드웨어의 주소를 직접적으로 사용했다면, 원하는 값이 나오는지 확인할 수 있는 방법이 없다. 따라서, 이와 같이 함수 고유의 동작에 관련을 가지고 있지 않은 외부 자원에 대한 접근을, 외부

에서 삽입하는 방식으로 의존성을 함수와 분리시킨 경우라고 볼 수 있다. 따라서, 이런 "의존성 분리"의 원칙은 코드의 재사용성을 높임과 더불어, 코드를 더 테스트하기 쉽게 만들어준다.

06. 함수가 다른 함수를 호출하는 것을 가능한 줄인다.

함수가 다른 함수를 많이 호출하면 할 수록 외부 모듈에 의존성이 강해진다. 함수는 될 수 있으면 외부에 의존적이지 않게 만들어야 테스트 하기도 편하다. 의존성이 늘어날수록 그 의존성을 대체할 수 있는 더 미(Dummy) 함수들을 많이 만들어야 한다. 외부에 대한 의존성이 단위 테스트를 어렵게 만드는 주된 이유이기도 하다. 일반적으로 기존에 이미 구현된 코드의 경우 이와 같은 의존성 때문에 단위 테스트를 포기하도록 만든다. 따라서, 가능한 적은 수의 외부 함수에 대한 의존성을 가져야 할 것이다.

```
int function_A( int x );
int function_B( int x );
int function_C( int x );
```

```
int function( int x ) {
    ...
    function_A( x );
    function_B( y );
    function_C( z );
    ...
    return xyz;
}
```

위의 경우 "function()"을 단위 테스트하기 위해서는 "function_A(), function_B(), function_C()"라는 세 개의 함수를 정의해 주어야 한다. 그렇지 않다면, "function()" 함수가 제대로 동작하지 않을 것이기 때문이다. 따라서, 각각의 호출되는 함수 들에 대한 입력과 출력을 조사해서 "function()" 함수가 요구하는 결과가 무엇인지 확인해야 한다. 이미 호출되는 함수들이 만들어진 경우라면, 그것을 활용할 수 있는 방법도 있겠지만, 없다면 가상으로라도 만들어서 단위 테스트를 만들어야 한다. 이를 위해서 목(Mock)을 활용하는 방법도 있다. 목(Mock)을 활용하는 경우에는 목(Mock)에서 호출되는 함수들이 올바르게 사용 되고 있는지 확인하는 방법도 같이 만들어야 한다.

07. 함수가 사용하는 전역변수를 없앤다.(혹은, 최소한으로 유지한다.)

함수가 전역변수를 사용하면 그 변수는 다양한 시스템의 상태를 반영할 가능성이 높다. 그런 변수들이 늘어나면, 테스트 전에 설정해 주어야 하는 경우의 수가 늘어나게 된다. 될 수 있으면 전역변수를 사용하지 않는 것이 최선이다. 전역변수는 의도하지 않은 변경을 만들 가능성이 높으며, 문제가 발생 했을 때 원인을 찾기 어렵게 만든다. 따라서, 만약 전역변수가 반드시 필요하다면, 차라리 전역변수를 접근하는 접근 함수(getter/setter)를 사용하는 것이 도움이 될 것이다.

```
int global_variable;

static inline int getter( void ) {
    return global_variable;
}

static inline setter( int x ) {
    global_variable = x;
}

int function( int x ) {
    int state = getter();
```

```

...
setter( x );
...
return xxx;
}

```

위의 함수는 “getter/setter”를 이용해서 전역변수를 사용하는 예를 보여준다. 인라인(inline)함수로 선언된 “getter/setter”는 단순히 값을 가져오거나 설정하는 역할을 하지만, 부가적인 역할도 할 수 있도록 만들 수 있다. 내부에서 전역변수를 사용하기에, 전역변수에 적절한 값이 설정되는 것을 알기 위해서는 전역변수에 직접적으로 접근하는 것을 간접적인(Indirect)방식으로 변경했다. 단위 테스트에서는 함수의 복귀값 만이 아니라, 전역변수에 접근하는 “getter/setter”에서도 정확한 값을 입력/출력하는지를 확인할 수 있게 되었다.

08. 함수에 중복된 코드를 없앤다.(혹은, 새로운 함수로 중복을 대체한다.)

중복된 코드는 항상 문제를 발생시킬 위험이 많은 부분이다(일반적으로 버그의 온상이라고 부름). 코드의 수정이 발생할 때, 관련된 코드들을 전부 같이 수정해 주어야 하지만, 그렇지 못해서 기존의 코드가 남아있을 가능성이 높다. 또한, 코드의 크기가 민감한 경우에도 악영향을 줄 수 있다. 물론, 실행 측면에서는 속도가 더 좋을 수도 있지만, 관리적인 측면을 더 많이 생각하는 것이 좋다. 원칙은 “하나의 수정이 필요한 이유에 대해서, 한 곳에서만 수정이 일어나야 하는 것”이다.

```

int function_A( void ) {
    ...
    /* 중복된 부분 */
    if ( xxx == yyy ) {
        do_something();
    }
    ...
    return zzz;
}

int function_B( void ) {
    ...
    /* 중복된 부분 */
    if ( xxx == yyy ) {
        do_something();
    }
    ...
    return kkk;
}

```

만약, 위와 같이 공통된 부분이 있는 코드라면, 중복된 코드를 뽑아내서 새로운 함수를 만든 것을 생각해보아야 한다. 즉, 다음과 같은 인라인 함수로 대체할 수 있다.

```

static inline void duplicated_code( void ) {
    if ( xxx == yyy ) {
        do_something();
    }
    return;
}

```

```

int function_A( void ) {
    ...
    duplicated_code();
    ...
    return zzz;
}

int function_B( void ) {
    ...
    duplicated_code();
    ...
    return kkk;
}

```

위에서는 반복된 코드 새로운 인라인 함수를 사용해서 분리(뽑아내서, "Extract Method")시켜서 관리하게 되었다. 나중에 코드를 수정할 경우, 복제된 코드가 한 곳에서 관리되기 때문에, 훨씬 수월하게 관리할 수 있다. 당연히 단위 테스트에서는 이와 같이 분리되어 있는 코드는 더 세밀하게 테스트 할 수 있는 기회를 제공할 것이며, 단위 테스트를 위한 스크립트 자체도 이와 같이 중복이 없이 관리되어야 할 것이다. 복제된 코드가 없다는 것은 그 만큼 관리하기 쉬운 코드와 테스트를 유지하는데 도움이 될 수 있다.

09. 함수를 될 수 있으면 전역 함수로 만들지 않는다.

사실 전역으로 선언되지 않은 함수는 테스트 코드와 분리하기 어렵다. 즉, 그와 같은 함수를 접근하기 위해서는 동일한 파일내에 단위 테스트를 위한 코드가 삽입되어야 한다. 이를 극복하기 위해서는 파일 전체를 포함("include")시켜서 단위 테스트 파일에서 직접적으로 접근할 수 있도록 만들어야 할 필요도 있다. 하지만, 더 적은 정보를 외부에 노출하기에 함수의 안정성은 더 높다고 볼 수 있다. 전역 함수들이 많다면, 그 만큼 테스트 해야할 경우의 수가 많다는 것을 의미하며, 오류에 대한 방어적인 코딩도 그 만큼 더 신경써야 한다. 내부적으로 사용될 함수들은 입력을 가정할 수 있기에, 테스트 케이스의 범위를 한정 시킬 가능성도 높다.

만약, 내부적으로 사용되는 함수들(Helper 함수를 포함해서)을 간단히 "_"와 같은 것을 정의해서 이름을 붙일수 있으며, 이런 내부적인 함수들은 간략한 기능이나, 기존의 기능과 연결고리 역할을 하는 래퍼(Wrapper)에서 호출 되도록 만들 수 있다.

```

static int _helper_function( int x ) {
    ...
    /* Do something in Old Version */
    ...
    return xxx;
}

int function( int x ) {
    ...
    if ( old_version ) {
        return _helper_function( x );
    }
    /* Do something in New Version */
    ...
    return xxx;
}

```

위의 경우에서 “Helper”함수로 사용된 것은 파일 내부에서만 접근 가능한 형태이며, 외부에서는 사용하지 못한다. 함수에서 새로운 버전을 확장하려고 하는 경우, 이미 만들어진 기존 코드(Legacy Code)에 대한 래퍼로 동일한 함수 API를 제공하고, 이를 새로운 버전과 이전 버전이 호환 되도록 만든 경우다. 따라서, 기존 코드를 단위 테스트를 이용해서 기능을 확장 하려는 경우에 활용해 볼 수 있다.

10. 함수가 특정 상태에 민감하지 않도록 만든다.

함수가 특정 상태를 가지는 것은 그 코드를 테스트하기 위해서 특정 상태를 만들어야 하는 부담이 따른다. 함수가 상태를 유지하기 위해서는 내부에 정적변수를 사용하거나, 파일 범위의 변수를 사용하는 것이며, 이때는 내부적으로 생성된 정적변수를 확인하는 새로운 함수가 필요하게 된다. 물론, 이 변수가 옳바른 값을 가지는 것을 확인하는 것과 함수의 복귀값을 확인하는 두 가지가 필요할 수 있다. 만약, 정적 변수의 값을 확인하는 코드가 사용 되지 않는 함수라면, 테스트 만을 위해서 그런 변수를 선언하는 것은 테스트 자체를 우회적으로 하도록 요구한다. 이러한 변수를 함수의 외부로 드러나게 만들어, 단위 테스트에서 해당 변수를 확인할 수 있도록 만들어줄 수도 있다.

```
int function( int x ) {
    static int k = 0;
    ...
    k = x;
    ...
    return xxx;
}
```

위와 같이 정의된 함수는 내부에 정의된 상태를 가지는 변수 "k"의 값을 사용한다. 즉, 함수의 외부에서는 "k"의 값을 확인할 수 있는 테스트 방법이 없다. 이를 개선하기 위해서 다음과 같이 변경하도록 한다.

```
static int k = 0;

int function( int x ) {
    ...
    k = x;
    ...
    return xxx;
}
```

위와 같이 변경하면, 단위 테스트에서는 변수 "k"의 값을 확인할 수 있게된다. 하지만, 문제는 "k"의 값을 파일내의 다른 함수 들도 접근할 수 있게 된다는 점이다. 즉, 같은 파일내의 다른 함수 들에서 의도하지 않은 접근을 할 수 있는 가능성이 높아진다. 따라서, 변수의 범위를 한정시켜서 사용해야하는 원칙에는 위배된다고 볼 수 있다. 또한, 특정 상태를 미리 설정해야 하고, 그 상태에 의존적인 테스트가 만들어지기기에 상태의 정의에 따라 테스트도 추가되어야 한다.

주로 병렬로 실행될 수 있는 코드는 정적 변수나 전역 변수에 대해서 동기화(Synchronization)문제를 일으킬 가능성이 있기에 만들지 않는 것이 좋다. 값을 내부에 저장하기 보다는 외부로 전달하는 것이 문제를 회피하는 한가지 방법이 될 수 있다. 그렇지 않다면, 테스트에서 동기화 까지도 포함할 수 있는 테스트 케이스를 만들어야 할 필요도 있을 것이다. 하지만, 동기화는 언제 실행 될지 알 수 없기에 테스트가 어려운 점이 있다는 것을 주의해야 한다.

[Unity를 이용한 C 단위 테스트]

단위 테스트는 가장 기본적인 실행 가능한 코드에 대한 테스트를 작성하고, 이를 실행하는 것을 말한다. 여기서, 말하는 “가장 기본적인 실행 단위”는 C언어에서 함수(Function)에 해당한다고 이미 이야기 했

었다. 즉, 함수 단위로 테스트를 작성해서, 제대로 함수들이 원하는 역할을 수행하는지 확인하는 작업이 단위 테스트다.

단위 테스트가 중요한 이유는, 복잡한 코드를 구성하는 기본 단위 들에 대해서 복잡도를 낮춰서 테스트를 만들어 검증할 수 있다는 부분이며, 복잡한 코드에서 문제를 찾기 보다 기본 단위에서부터 검증을 통해 품질이 높은 코드를 만들어낼 수 있다는 점이다. 따라서, 소프트웨어 개발자가 가장 친근하게 사용할 수 있는 도구로 항상 코드의 개발과 같이 병행되어야 할 일상적인 개발 활동에 포함된다.

대부분의 개발자들이 자신의 코드를 테스트하는데 익숙하다고 생각하지만, 다른 사람의 코드와 결합될 경우에는 문제를 찾아내는데 오랜 시간이 걸리는 것이 일반적이다. 이때, 자신의 코드에 문제가 없다는 것을 증명할 수 있다면 어떨까? 그럴 수만 있다면, 좀더 버그를 찾아내는 활동에 있어서 훨씬 효과적으로 대응할 수 있다(물론, 그렇다고 남을 비난하기 위한 도구로 테스트를 사용하는 것은 아니지만).

문제는 개발자들이 단위 테스트를 하지 않는다는 점이며, 단위 테스트를 만들기 위한 시간을 아깝다고 생각하는 것이다. 하지만, 정말 중요한 것은 단위 테스트는 품질을 높이기 위한 소프트웨어 개발과 관련된 가장 기초적인 역량이라는 점이다. 그외에 코드의 품질을 높일 수 있는 방법으로는 다양한 코드 리뷰 방법들이 있다.

단위 테스트가 잘 되지 않는 상황에서는 다른 자동화를 이용한 품질 확보 노력이 한정된 수준에 그치고 말 것이다. 단위 테스트를 확대하면 통합 테스트도 가능하게 되며, 전체적으로 기능에 대한 검증도 구현이 완료되는 순간 바로 확인할 수 있게된다. 따라서, 어떤 일을 했다는 종료 조건(Exit Criteria, "Done, Done", "Definition of Done:DoD")으로도 단위 테스트는 중요한 부분이다. 따라서, 소프트웨어 개발자라면 반드시 이를 잘 활용할 능력을 갖추어야 할 것이다.

여기서, 사용할 단위 테스트 도구는 Unity와 CppUTest, GTest 등이다. 이들 각각은 C언어와 C++언어를 위해서 만들어진 테스트 프레임워크 이지만, 둘다 C언어의 함수 단위의 테스트를 할 수 있도록 지원한다. 그리고, Eclipse와 같은 IDE(Integrated Development Environment)를 활용해서 테스트의 커버리지(Test Coverage)를 측정할 수도 있다. 먼저, Unity 환경을 구축하기 위해서 다음과 같은 것을 설치해야 한다.

- Eclipse CDT : C/C++을 개발하기 위한 IDE 환경을 제공하는 소프트웨어
- Unity : C언어를 위한 Unit Test Framework 소프트웨어
- Linux Tools : Eclipse CDT에서 Plugin으로 사용되는 소프트웨어(Gcov, GProf)
- LCOV : 코드의 테스트 Coverage를 측정해서 Web으로 보여주기 위한 소프트웨어
- CMock : Unity를 위해서 사용할 목(Mock)

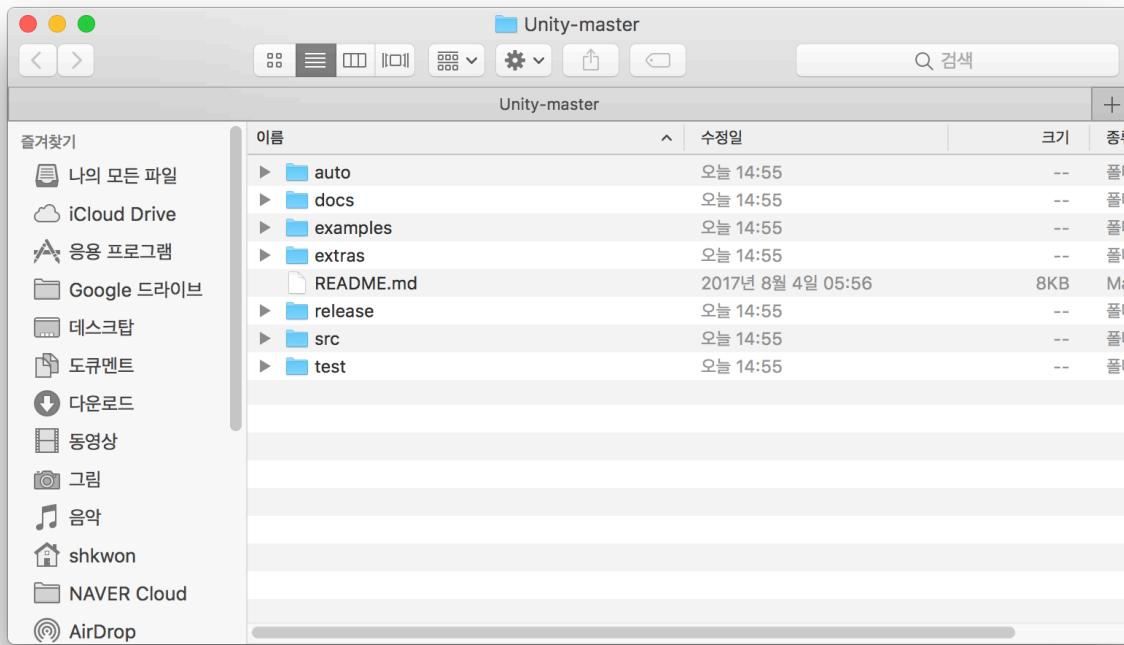
물론, 이 밖에도 다른 소프트웨어들이 필요하지만, 이 정도만 가지고 시작하도록 하겠다. 더 필요한 것들이 생긴다면, 그때가서 다시 추가하도록 하겠다. 먼저, Unity를 다운받아서 적절한 디렉토리에 압축을 해제하도록 한다.

1. “Unity” 테스트 프레임워크(Test Framework)의 다운로드

“Unity”는 C언어 전용 단위 테스트로 사용할 수 있는 프레임워크다. 게임을 개발하는 사람에게는 다른 “Unity”가 있기에, 혼동 해서는 안된다. 단위 테스트를 위한 “Unity”는 아래에서 다운로드 받을 수 있다.

<http://www.throwtheswitch.org/unity/> 또는, <http://sourceforge.net/projects/unity>

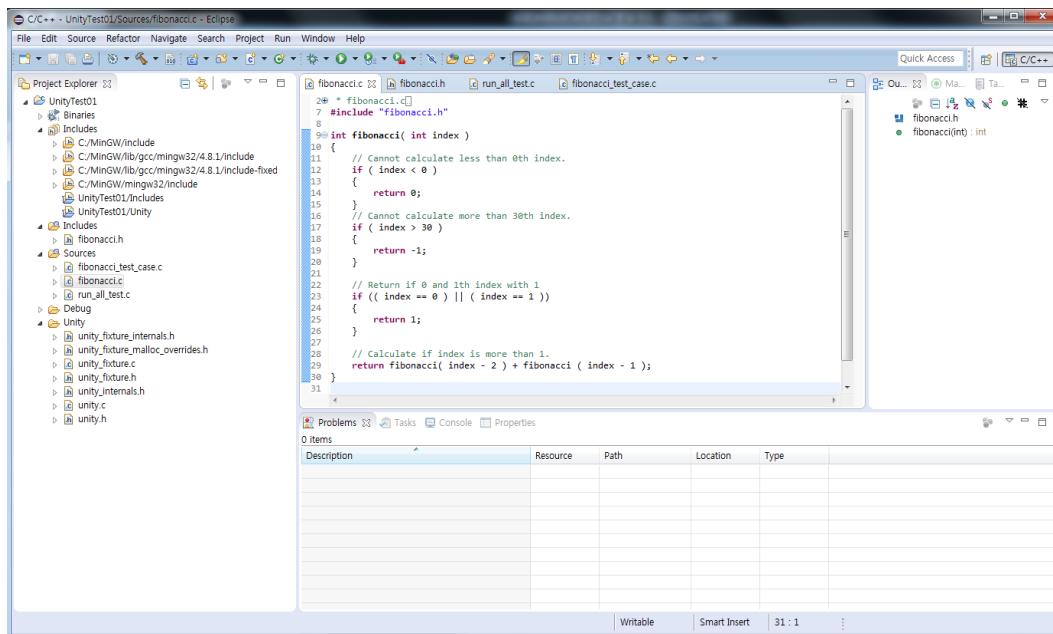
다운로드를 받으면 압축된 파일로 되어 있을 것이다. 이를 그냥 원하는 디렉토리에 압축을 해제해 주면 된다. 아래 그림은 해제된 “Unity”의 디렉토리 구조를 보여준다.



Unity를 압축해제하면 여러가지 디렉토리가 볼 수 있지만, 이중에서 사용하고자하는 것은 "src" 아 "extras(fixture/src)"에 있는 파일 들이다. Eclipse에서 프로젝트를 새로 만들어서 필요한 파일들을 불러들이도록 한다.

2. “Unity”의 프로젝트 추가

방법은 파일을 프로젝트에 추가하면 된다. 새로운 프로젝트를 Eclipse에서 만들고, "File -> New -> Source Folder"를 이용해서 소스코드를 넣을 디렉토리를 생성하고, "File -> New -> Source File"을 이용해서 파일들을 추가한다. 이때, 각종 헤더 파일이나 소스 파일이 필요하기 때문에, 프로젝트의 “C Compiler” 옵션의 설정을 변경할 수 있다. 예를 들어, 헤더 파일들을 포함시키기 위해서 “-I”와 같은 옵션을 추가해 줄수도 있다.



3. 테스트 함수의 작성

일단 “Unity”를 포함한 프로젝트의 설정이 마무리 되었다면, 이제는 테스트 대상이 되는 함수들을 만들어 주도록 한다. 일반적으로 이미 테스트 해야할 함수가 만들어져 있는 경우가 많기에, 한번에 여러 개의 함수를 테스트 해야할지도 모른다. 여기서는 단위 테스트를 단순화 시키기 위해서 함수를 만들면서 테스트 하는 방법을 취하도록 하겠다.

```
/* fibonacci.c */
#include "fibonacci.h"

int fibonacci( int index ) {
    if ( index < 0 )
    {
        return 0;
    }
    if ( index > 30 )
    {
        return -1;
    }

    if (( index == 0 ) || ( index == 1 ))
    {
        return 1;
    }

    return fibonacci( index - 2 ) + fibonacci ( index - 1 );
}

/*fibonacci.h */
#ifndef FIBONACCI_H_
#define FIBONACCI_H_

int fibonacci( int index );

#endif /* FIBONACCI_H_ */
```

4. 테스트 케이스 실행의 준비

Unity의 코드 중에서 7개의 파일이 필요하다. “unity.c, unity.h, unity_fixture.c, unity_fixture.h, unity_internals.h, unity_fixture_internals.h, unity_fixture_malloc_overrides.h” 파일들을 포함시켰다. 이제는 Unity 테스트를 실행하기 위해서 필요한 “main()”함수를 정의할 파일을 만들도록 한다. 내용은 아래와 같은 것이 필요하다.

```
#include "unity_fixture.h"

static void RunAllTests(void) { /* 모든 테스트 케이스의 모음을 실행 */
    RUN_TEST_GROUP(Fibonacci);
    /* 추가할 테스트 케이스 그룹은 이곳에 추가할 수 있다. */
}

int main(int argc, char * argv[]) {
```

```
    return UnityMain(argc, argv, RunAllTests); /* 테스트 케이스 그룹 들을 실행한다. */
}
```

"unity_fixture.h"는 Unity Test Framework을 사용하기 위해서 필요하며, "RunAllTests()" 함수는 모든 정의된 테스트 그룹을 실행하기 위해서 필요하다. "UnityMain()" 함수는 "RunAllTests()"를 실행시켜준다. 위의 코드에서는 테스트 그룹으로는 "Fibonacci"로 정했다. 나중에 테스트 그룹을 선언할 때 이것을 사용하도록 해야 한다.

5. 테스트 케이스의 작성

이제는 테스트 케이스(Test Case)를 생성할 차례이다. 이때, 각각의 테스트 케이스들은 같은 종류를 묶어서 하나의 테스트 그룹으로 묶어주는 것이 좋다. 즉, 테스트를 관련이 있는 테스트를 분류하기 위해서 사용한다. 따라서, 앞에서 정의한 "Fibonacci"를 테스트 그룹으로 선언해서 사용하도록 하겠다.

```
#include <stdio.h>
#include "unity_fixture.h"
#include "fibonacci.h"

TEST_GROUP( Fibonacci ); /* 테스트 그룹 선언 */

TEST_SETUP( Fibonacci ) { /* 항상 테스트 케이스 실행 전에 실행 : 테스트 실행 준비 */
    printf("Setup Called!!!\n");
}

TEST_TEAR_DOWN( Fibonacci ) { /* 항상 테스트 케이스 실행 후에 실행 : 복구 */
    printf("Teardown Called!!!\n");
}

TEST( Fibonacci, MinusIndexTest ) { /* 테스트 케이스 */
    TEST_ASSERT_EQUAL( 0, fibonacci( -1 ) );
}

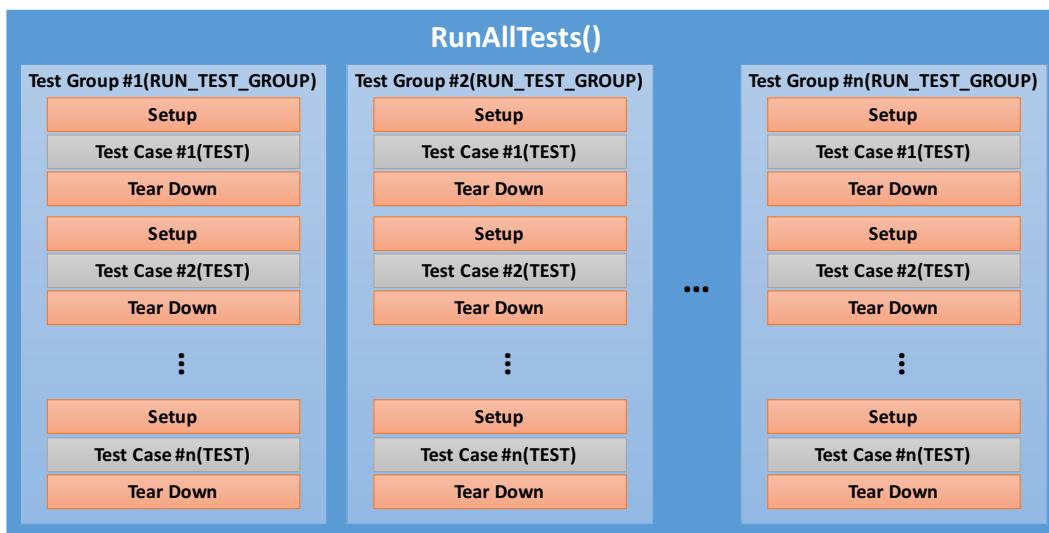
TEST( Fibonacci, MoreThanZeroAndSequenceTest ) { /* 테스트 케이스 */
    TEST_ASSERT_EQUAL( 1, fibonacci( 0 ) );
    TEST_ASSERT_EQUAL( 1, fibonacci( 1 ) );
    TEST_ASSERT_EQUAL( 2, fibonacci( 2 ) );
    TEST_ASSERT_EQUAL( 3, fibonacci( 3 ) );
    TEST_ASSERT_EQUAL( 5, fibonacci( 4 ) );
    TEST_ASSERT_EQUAL( 8, fibonacci( 5 ) );
    TEST_ASSERT_EQUAL( 13, fibonacci( 6 ) );
    TEST_ASSERT_EQUAL( 21, fibonacci( 7 ) );
    TEST_ASSERT_EQUAL( 34, fibonacci( 8 ) );
    TEST_ASSERT_EQUAL( 55, fibonacci( 9 ) );
    TEST_ASSERT_EQUAL( 89, fibonacci( 10 ) );
}

TEST( Fibonacci, MoreThan30Test ) { /* 테스트 케이스 */
    TEST_ASSERT_NOT_EQUAL( -1, fibonacci( 31 ) );
}
```

```
TEST_GROUP_RUNNER( Fibonacci ) { /* 테스트 케이스 그룹의 실행 */
    RUN_TEST_CASE( Fibonacci, MinusIndexTest ); /* 테스트 케이스들 */
    RUN_TEST_CASE( Fibonacci, MoreThanZeroAndSequenceTest );
    RUN_TEST_CASE( Fibonacci, MoreThan30Test );
}
```

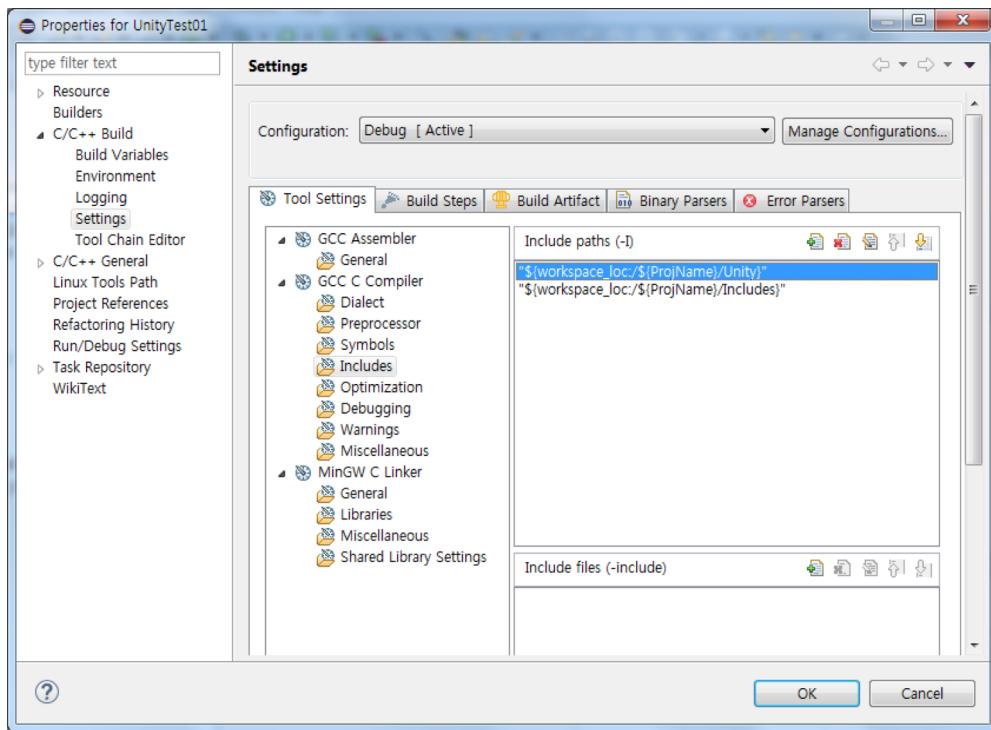
테스트 케이스를 실행하기 위해서는 "TEST_GROUP_RUNNER()"가 필요하다. 테스트 케이스 그룹에 속한 각각의 테스트 케이스에 대해서 "RUN_TEST_CASE()"를 사용해야 한다. 각각의 테스트 케이스는 "TEST_SETUP()"과 "TEST_TEAR_DOWN()"의 사이에서 실행된다. 즉, 하나의 테스트 케이스를 실행하기 전에, "TEST_SETUP()"이 실행되고, 테스트 케이스가 실행된다. 테스트 케이스의 실행 후에는 "TEST_TEAR_DOWN()"이 실행된다.

따라서, "TEST_SETUP()"에서는 테스트 케이스의 실행을 위해서 필요한 선행조건을 만들어 주고, "TEST_TEAR_DOWN()"에서는 "TEST_SETUP()"에서 만들어준 선행조건을 원래의 상태로 복구해 주면 된다. 예를 들어, 테스트를 위해서 필요한 메모리 공간의 할당이 있었다면, 나중에 해제하기 위해서 사용할 수 있다. 이는 각각의 테스트 케이스가 독립적으로 실행되도록 만들기 위한 환경을 제공하기 위해서 필요하다.



여기까지 했다면, 이제는 컴파일 및 실행이 가능할 것이다. 만약, 컴파일시 “Unity”관련 오류가 발생한다면, “Unity”가 필요한 헤더 파일들을 찾을 수 있도록 프로젝트 “Properties”를 변경해 주어야 한다. 설정은 아래에서 확인 하도록 한다.

“Project->Properties->C/C++ Build->Settings->Tool Settings->[C] Compiler”的 “include”에서 필요한 디렉토리를 추가하면 된다.



6. 단위 테스트의 실행

실행한 결과는 Eclipse의 “Console”에서 확인할 수 있다. 실행한 결과는 각각의 단위 테스트에 대한 실행 결과와 전체 단위 테스트 실행에 대한 결과를 아래와 같이 보여줄 것이다.

Unity test run 1 of 1

.Setup Called!!!

Teardown Called!!!

.Setup Called!!!

Teardown Called!!!

.Setup Called!!!

..\Sources\fibonacci_test_case.c:45:TEST(Fibonacci, MoreThan30Test):FAIL: Expected
Not-Equal

Teardown Called!!!

3 Tests 1 Failures 0 Ignored

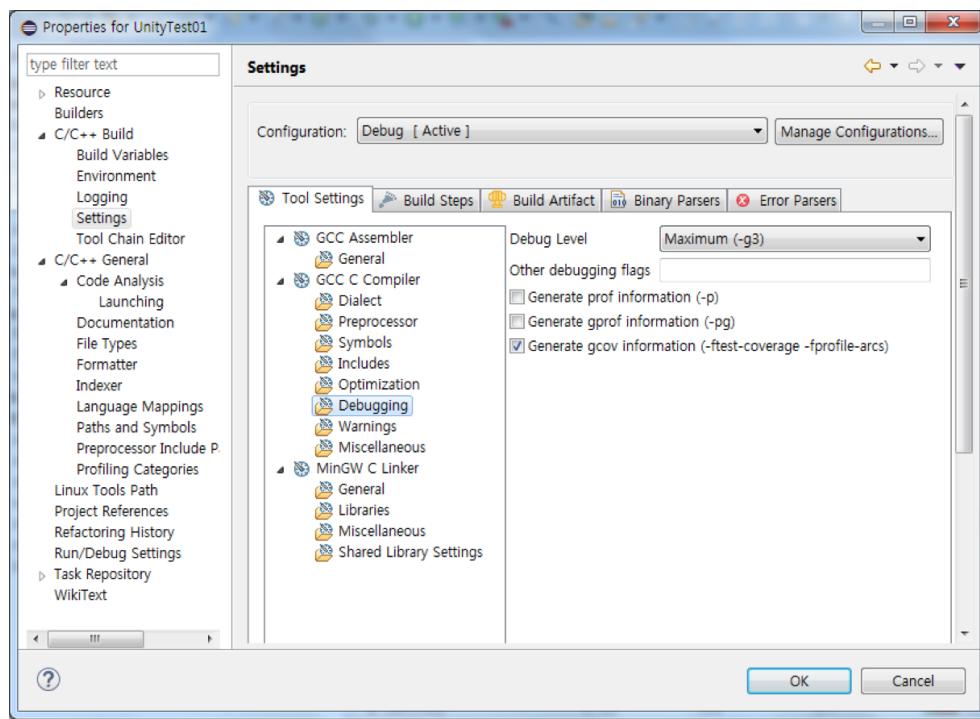
FAIL

각각의 테스트가 실행되기 전에 "TEST_SETUP()"이 호출되고, 테스트 케이스의 실행이후에 "TEST_TEAR_DOWN()"이 호출 되었다는 것을 확인할 수 있을 것이다. 여기서는 3개의 테스트 케이스가 실행이 되었으며, 1개의 테스트 케이스(MoreThan30Test)가 실패했다는 것을 볼 수 있다. 즉, "-1"이라는 값과 달라야 한다고 테스트 케이스를 만들었지만, 실제로는 같은 값을 "fibonacci()"함수가 돌려주었기 때문에 실패라고 판단한 것이다(물론, 코드에는 문제가 없다. 단지 실패하는 테스트 케이스를 만들기 위해서 만들었을 뿐이다.). 이를 통해서 "fibonacci()"함수가 의도 했던대로 실행된다는 사실을 확인할 수 있다.

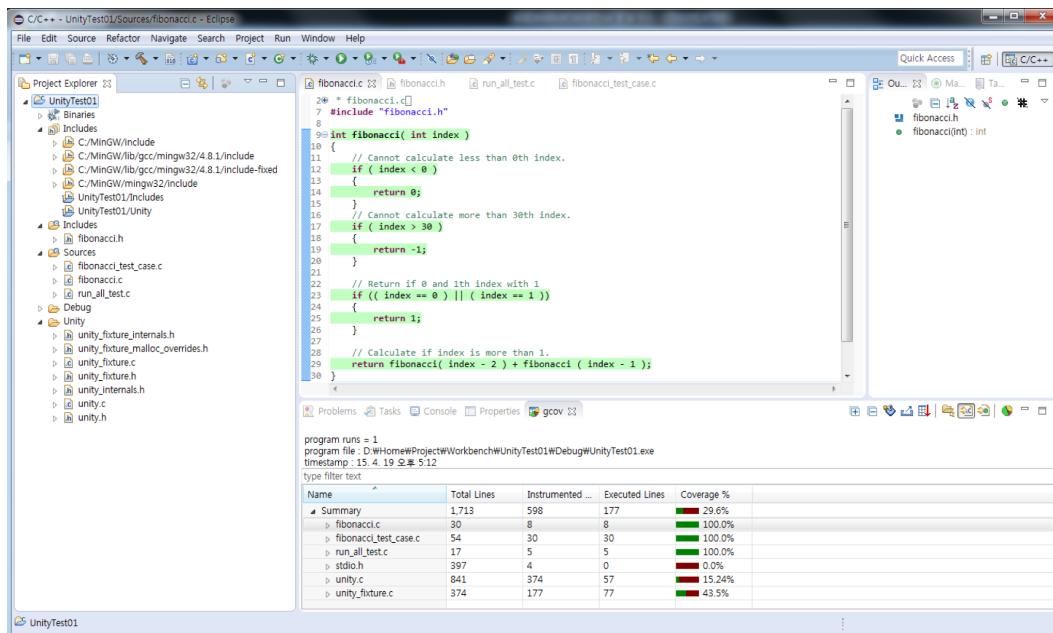
7. 테스트 커버리지(Coverage) 확인하기

테스트 케이스가 정말로 내가 만든 함수를 잘 테스트할 수 있다는 것을 알 수 있는 방법은, 함수의 어떤 부분이 테스트 케이스를 이용해서 테스트 되었는지 눈으로 확인하는 방법이 가장 좋다. 이를 위해서는 "GCOV(GNU Coverage)"라는 툴을 사용하도록 한다. 실행 정보를 기억하기 위해서, 프로젝트 설정에서 "Profiling Tools -> Profile Code Coverage"를 선택하도록 한

다. 선택과 동시에 컴파일 설정을 변경 하라는 메시지가 나올 수 있으며, 이를 없애기 위해서는 프로젝트의 빌드(Build) 설정을 변경한다.



"-fprofile-arcs"가 설정되도록 "Debugging"옵션에서 체크박스를 선택하도록 한다. 필요하다면, 다른 나머지 체크박스들도 같이 설정할 수 있다. 이제는 실행한 후에 실행 결과를 확인할 차례다. Eclipse의 하단에 있는 "gcov"에서 확인할 수 있다.



그림에서 보듯이 각각의 파일 별로 얼마나 많은 코드들이 실행 되었는지 "%"로 확인할 수 있으며, 실행된 코드에 대해서는 녹색으로 표시되어 있다. 실행되지 않은 코드들은 붉은색으로 표시된다. 즉, 이와 같은 결과를 가지고 실행되지 않은 부분에 대해서, 실행할 수 있는 테스트 케이스를 추가하면 될 것이다.

간혹, 실행은 되었지만 붉은 색으로 표시된 코드들을 볼 수도 있을 것이다. 대부분 크게 우려할 부분은 아니기에, 자신이 작성하고 있는 코드에 집중된 커버리지만 주의하면 된다. 실행되지 않은 코드를 테스트 하기 위해서는 추가적인 테스트 케이스를 만들거나, 실행하기 위한 선행 조건을 일부 수정해야 할 수도 있다. 만약, 선행 조건이 모든 테스트 케이스에 대해서 공통적으로 사용된다면, “TEST_SETUP()”에서 처리하면 되고, 각각의 테스트 케이스마다 다르다면 해당 테스트 케이스에서 먼저 설정해 주어야 할 것이다.

[CMock을 이용한 단위 테스트]

단위 테스트를 하고 있는 함수가 다른 함수를 호출하는 의존성을 가지고 있다면, 호출되는 함수를 테스트를 하고 있는 함수와 분리해 주어야 한다. 즉, 테스트 중인 코드 만이 검증 대상이지, 그것이 의존하고 있는 함수 들까지 모두다 테스트 할 필요는 없다(물론, 다른 함수가 이미 작성 되어 있다면, 같이 테스트 할 수도 있다. 하지만, 이 경우에는 테스트 범위가 커지기에, 문제 발생시 분석해야 할 범위가 늘어날 가능성이 있다). 이 경우 테스트 대상이 되는 함수가 의존하고 있는 함수 들을 다른 것으로 대체해 주어야 하는데, 이때 사용 되는 것이 테스트 더블(Test Double)의 일종인 목(Mock)이다.

C언어에서 단위 테스트 시에 사용할 수 있는 목(Mock)으로는 다양한 것들이 있겠지만, 대표적으로 “CMock”과 “CppUMock”을 찾을 수 있다. “GoogleMock”과 같은 것을 이용해서 할 수도 있지만, 여기서는 일단 “CMock”을 이용해서 어떻게 단위 테스트를 하는지 보도록 하겠다. “CMock”的 경우에는 헤더 파일을 해석해서 필요한 목(Mock) 함수 들을 자동으로 생성하는 기능이 있으며, 그렇게 생성된 코드를 이용해서 테스트하고 있는 함수가 “예상된(Expected)값”을 제대로 사용하고 있는지 확인하는 방식으로 동작한다.

1. CMock의 설치

“CMock”을 사용하기 위해서는 “Ruby”라는 스크립트 언어를 이용해야 한다. 그리고, “CMock”을 다운로드 받기 위해서 “GitHub”접근을 위한 “Git”을 설치해야 한다. “Git”을 설치하는 이유는 단순히 “Zip”으로 만들어진 파일을 받으면, “CMock”을 위한 파일 들만 받을 것이기 때문이다. “CMock”에서 관리되는 “GitHub”에는 “Unity”도 같이 포함되어 있다. 따라서, 이런 것들을 호환성에 맞춰서 받기 위해선 “Git”을 사용해서 “CMock”을 받도록 하자.

```
> git clone --recursive https://github.com/throwtheswitch/cmock.git
```

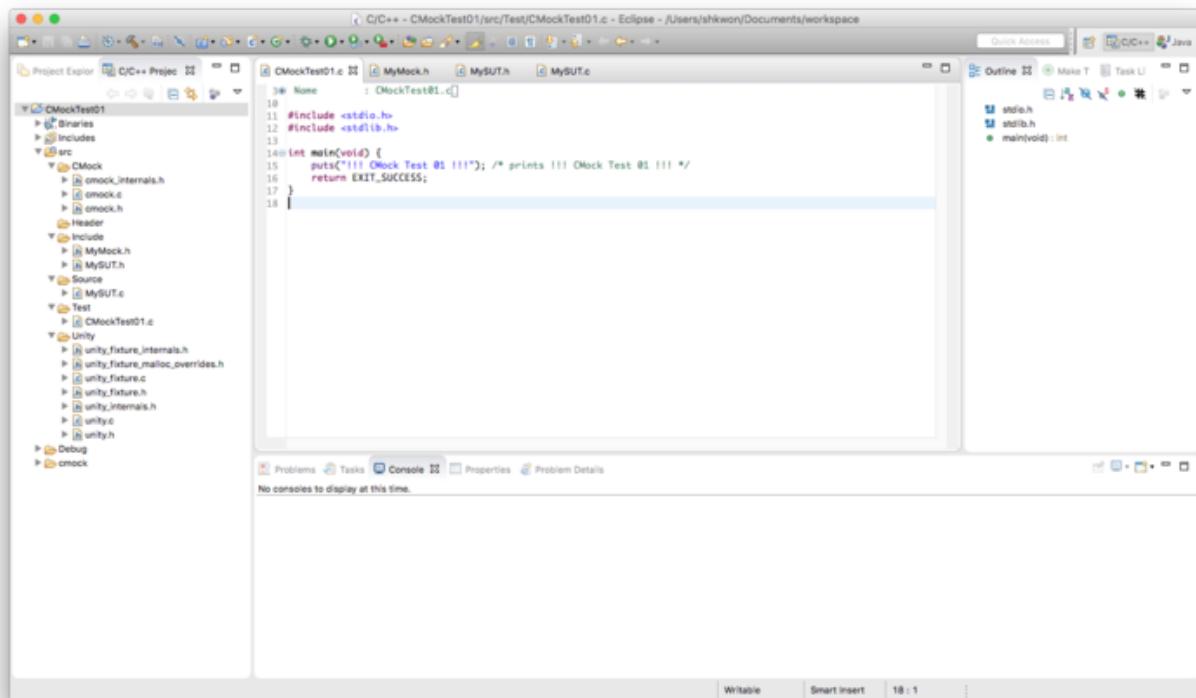
위와 같이 명령행(Command Line)에서 “Git” 명령어를 통해서 “CMock”을 다운받을 수 있지만, 윈도우 버전의 “Git”을 다운 받았다면, 앞의 명령어에서 “<https://...>”부분을 입력으로 명시해서 다운로드 받을 수도 있다. 맥(Mac)을 사용하는 사용자들은 “Git”이나 “Ruby”가 이미 설치 되어 있으므로, 따로 다운로드 받아서 설치할 필요는 없다. 다만, GUI방식이 아니기에, 명령행에서 실행하면 된다.

“CMock”을 다운받으면, “cmock/vendor” 디렉토리에 이미 “Unity”가 있기에, 단위 테스트에서 “Unity” 프레임워크를 그대로 사용하면 된다. 나중에 “CMock”이나 “Unity”관련 파일들이 컴파일 시에 필요하기에, “src”나 “extra/src”등과 같은 디렉토리에서 찾으면 될 것이다. “Unity”를 실행하기 위해서는 “main()”함수와 “테스트 그룹”등을 만들어야 하며, 실제로 어떻게 하는지를 다음에서 확인 할 것이다.

각주

2. 프로젝트의 생성과 디렉토리의 구성

테스트를 위해서는 테스트 케이스를 기술하는 코드와 개발하고 있는 소스코드, 테스트 프레임워크 및 “CMock”등이 분리된 디렉토리로 관리하는 것이 좋다. 같은 디렉토리에 있다면 나중에 테스트를 추가하거나 코드를 수정하는 등의 일을 복잡하게 만들 것이다. 아래의 그림은 Eclipse를 이용해서 프로젝트를 생성한 후 각각의 디렉토리를 만들어준 경우다.



그림에서는 아직 “Unity”를 이용해서 단위 테스트를 하기 위한 스크립트를 만든 상태는 아니다. 이후에 계속 “Unity”를 이용해서 단위 테스트를 진행하는 과정을 설명해 나가도록 하겠다. 일단은 이 정도까지 구축 했다면, 단위 테스트 및 “CMock”을 사용할 준비는 마친 것이다.

3. CMock을 이용한 Mock Object의 생성

“CMock”에서 제공하는 “Ruby” 스크립트는 테스트 하고자 하는 코드가 의존성을 가진 함수들을 목(Mock)으로 자동 생성해 준다. 앞의 그림에서는 이미 목(Mock)을 생성 했으며, 아래와 같은 명령어를 이용했다.

```
> ruby cmock.rb --mock_path="mock_path_name" myfile.h
```

여기서, "cmock.rb"는 Mock을 생성하기 위해서 호출되는 "Ruby" 스크립트이며, "--mock_path" 생성될 목(Mock)이 위치할 디렉토리를 표현한다. "myfile.h"는 목(Mock)으로 대체하고자 하는 "Production Code"의 헤더 파일이다.

생성된 목(Mock) 오브젝트(여기서는 생성된 목 그 자체를 가리키는 말로 쓴다)를 프로젝트의 적절한 디렉토리로 배치할 필요가 있다. 기본적으로 목(Mock)은 생성 시 헤더 파일과 구현(.C) 파일이 같이 나온다. 헤더 파일은 테스트를 작성하기 위해서, 구현 파일은 목(Mock)의 실제 구현을 담고 있기에 테스트 해야 할 코드(Production 코드: 생산에 사용될 코드)에서 호출하기 위해서 필요하다.

[주의]

1. 인터넷에서 찾은 “CMock”파일의 경우 “Unity”가 포함 안된 경우가 있을 수 있기에, 그럴 경우에 는 다운로드 받은 “Unity”를 그대로 “CMock/vendor/unity”에 복사해서 사용해도 된다.
2. “ruby” 명령어 실행이 안될 경우에는 “ruby”에 대한 환경 설정이 되어 있는지 확인해야 한다. “cmock.rb”的 실행에 문제가 발생하면, 다운로드 받은 “CMOCK”에 “Unity”가 포함되어 있는지 확인해야 한다. 또한, 디렉토리나 파일에 대해서도 경로를 제대로 쓰고 있는지 확인할 필요가

있다. 아무런 내용이 생성되지 않고 오류 메시지도 없다면, 생성하고자 하는 목(Mock)에 대한 입력파일을 찾지 못한 경우이며, 이때는 절대 경로를 파일의 이름과 같이 명시해야 한다.

4. “Unity” 프레임워크를 이용한 단위 테스트 준비

이제는 단위 테스트를 위해서 “Unity”를 실행할 준비를 해야한다. 단위 테스트를 위한 "main()"함수가 정의된 파일과 단위 테스트를 가지고 있는 파일을 아래와 같이 준비하도록 하자. 이와같이 나누는 이유는 테스트 케이스 들에서 공통적으로 사용 되는 "Setup"을 단위 테스트 그룹별로 파일을 나누어 줄 수 있기 때문이다.

```
#include <stdio.h>
#include <stdlib.h>

#include "unity_fixture.h"

static void RunAllTests( void ) {
    RUN_TEST_GROUP( CMockTestGroup );
}

int main( int argc, const char *argv[] ) {
    return UnityMain( argc, argv, RunAllTests );
}
```

위의 코드는 "main()"함수에서 "UnityMain()"함수를 호출하도록 했고, 그곳에서 모든 테스트 그룹을 실행하는 "RunAllTests()"를 호출했다. 각각의 테스트 케이스가 실행되는 그룹은 "CMockTestGroup"과 같은 이름을 가진다. 아래는 테스트 그룹을 정의한 파일을 보여준다.

```
#include "unity_fixture.h"
#include "MockMyMock.h" /* 생성된 Mock의 헤더 파일 */
#include "MySUT.h"

TEST_GROUP( CMockTestGroup ); /* 테스트 그룹의 생성 */

TEST_SETUP( CMockTestGroup ) { /* 테스트 그룹에 속한 테스트 케이스를 위한 설정 */
    myInfo.name = "SH Kwon";
    myInfo.age = 44;
    myInfo.alive= true;
}

TEST_TEAR_DOWN( CMockTestGroup ) { /* 테스트 그룹에 속한 테스트 케이스의 설정 해지 */
    /* TEST_FAIL_MESSAGE( "Test Start!!!!"); */ /* 메시지 출력용 */
    myInfo.name = "";
    myInfo.age = 0;
    myInfo.alive= false;
}

TEST( CMockTestGroup, changeInformationTestSuccess ) { /* 테스트 케이스 */
    INFO newInfo;

    newInfo.name = "MJ Yoon"; /* 테스트 케이스를 위한 데이터 생성 */
    newInfo.age = 44;
    newInfo.alive= true;
```

```

        TEST_ASSERT_EQUAL_INT( true, change_information( &newInfo ) );
    }

TEST_GROUP_RUNNER( CMockTestGroup ) {
    RUN_TEST_CASE( CMockTestGroup, changeInformationTestSuccess );
}

```

테스트 그룹에는 현재 하나의 테스트 케이스만 정의되어 있다("changeInformationTestSuccess"). "TEST_SETUP()"과 "TEST_TEAR_DOWN()"은 정의된 각각의 테스트 케이스마다 실행될 것이다. 따라서, 각각의 테스트 케이스에 대해서 공통적으로 실행할 일을 처리한다. "TEST_SETUP()"은 테스트 케이스를 실행하기 위한 전제 조건(Pre-condition)을 만들어주고, "TEST_TEAR_DOWN()"은 "TEST_SETUP()"에서 해준 일을 원래 상태로 되돌린다. 위의 코드에서는 단순히 구조체의 값을 초기화시키고, 다시 비우는 과정을 처리 했을 뿐이다.

"TEST_GROUP_RUNNER()"에는 현재 하나의 테스트 케이스만 실행되도록 했기에, 나중에 추가되는 테스트 케이스가 있다면 "RUN_TEST_CASE()"로 테스트 케이스를 추가하면 된다. 현재 단위 테스트의 대상이 되고 있는 함수는 "change_information()"이라는 함수이며, 이 함수가 "true"를 돌려줄 경우 테스트 케이스가 성공적이라고 가정했다. 즉, "TEST_ASSERT_EQUAL_INT()"를 이용해서, 함수의 복귀값과 주어진 값을 비교해서 동일한지 검사했다. 단위 테스트의 대상이 되는 함수를 호출하기 위해서 필요한 변수를 정의 했으며("newInfo"), 여기서 부터 목(Mock)을 이용한 테스트가 시작된다고 생각할 수 있다. 실제 구현된 코드를 잠시 보도록 하자.

```

/* Include_myMock.h */
#ifndef INCLUDE_MYMOCK_H_
#define INCLUDE_MYMOCK_H_

#include <stdbool.h>

typedef struct PersonallInfo {
    char *name;
    int age;
    bool alive;
}INFO;

INFO *get_information(); /* Mock으로 대체될 함수들 : 구현된 함수가 아니다. */
INFO *set_information( INFO *info );
void print_information( INFO *info );

#endif /* INCLUDE_MYMOCK_H_ */

/* MyMock.h */
#ifndef INCLUDE_MYSUT_H_
#define INCLUDE_MYSUT_H_

#include "MyMock.h"

extern INFO myInfo;

bool change_information( INFO *info ); /* 현재 단위 테스트 대상이 되는 함수 */

```

```
#endif /* INCLUDE_MYSUT_H */
```

"MyMock.h"파일은 목(Mock)으로 생성될 파일의 원본에 해당한다. 즉, 여기서 제공되는 자료형과 함수들이 목(Mock)으로 변환된 함수에서 사용될 것이다. 목(Mock)을 만드는 이유가 원래의 의존성이 있는 코드를 테스트 하고자하는 함수에서 분리하기 위한 방법이기에, 목(Mock)으로 자동 생성되는 함수는 간단히 사용할 수 있는 축소된 함수 정도로 생각해 볼 수 있다. 이때, 이렇게 축소된 함수를 사용하기 위해서, 테스트 케이스를 실행하기에 앞서 목(Mock) 함수들이 호출될 수 있는 파라미터 값과 그에 대응한 복귀 값을 미리 정해 주어야 한다. 이것들은 목(Mock)으로 생성된 다음과 같은 함수들이 담당한다.

```
/* 생성된 Mock의 헤더 파일 */
/* mockmymock.h */
#ifndef _MOCKMYMOCK_H
#define _MOCKMYMOCK_H

#include "MyMock.h"

/* Ignore the following warnings, since we are copying code */
#if defined(__GNUC__) && !defined(__ICC)
#if !defined(__clang__)
#pragma GCC diagnostic ignored "-Wpragmas"
#endif
#pragma GCC diagnostic ignored "-Wunknown-pragmas"
#pragma GCC diagnostic ignored "-Wduplicate-decl-specifier"
#endif

void MockMyMock_Init(void);
void MockMyMock_Destroy(void);
void MockMyMock_Verify(void);

/* Mock 역할을 해주는 매크로 정의와 함수 정의 */
#define define_get_information_ExpectAndReturn(cmock_retval)
get_information_CMockExpectAndReturn(__LINE__, cmock_retval)
void get_information_CMockExpectAndReturn(UNITY_LINE_TYPE cmock_line, INFO* cmock_to_return);
#define define_set_information_ExpectAndReturn(info, cmock_retval)
set_information_CMockExpectAndReturn(__LINE__, info, cmock_retval)
void set_information_CMockExpectAndReturn(UNITY_LINE_TYPE cmock_line, INFO* info, INFO* cmock_to_return);
#define print_information_Expect(info) print_information_CMockExpect(__LINE__, info)
void print_information_CMockExpect(UNITY_LINE_TYPE cmock_line, INFO* info);

#endif
```

위의 파일은 자동으로 생성된 목(Mock)의 헤더 파일이다. 실제 구현 파일 및 헤더 파일들은 자동으로 생성되었기에 수정할 필요가 없다. 여기서는 어떻게 미리 호출될 함수의 복귀 값을 정해주는지 보도록 하자. 코드의 아래 부분에 "#define ExpectAndReturn()"이나, "#define ... Expect()"가 그 역할에 해당하는 매크로 들이다.

앞에서 만들어진 코드들을 컴파일 후, 실행하면 오류 메시지가 발생할 것이다. 즉, 아직 우리는 예측값 (Expected Value)을 테스트 케이스에서 설정해 주지 않았다. 아래와 같은 테스트 실패가 발생했다면,

제대로 되고 있다는 뜻이다. 나중에 이를 성공적으로 바꾸는 방법에 대해서 살펴볼 것이다.

Unity test run 1 of 1

```
... /src/Test/CMockTestCase.c:29:TEST(CMockTestGroup,
changeInformationTestSuccess):FAIL:Function get_information. Called more times than
expected.
```

1 Tests 1 Failures 0 Ignored

FAIL

위의 예에서 보듯이 “get_information()”이라는 함수가 예측된(Expected) 것보다 “change_information()”함수에서 더 호출되었다는 테스트 실패 메시지를 볼 수 있다. 즉, 우리는 “get_information()”이 어떻게 호출 될지 아직 아무것도 정해준 것이 아니기에 당연히 오류가 발생한 것이다. 이제는 이것을 성공적으로 호출되도록 테스트 케이스를 다음에서 변경해 보겠다.

5. 테스트 케이스 만들기

_mock은 단순한 리턴 값 만을 돌려주는 더미(Dummy)와 같은 역할보다는 순서를 가진 시나리오를 검증하는데 적합하다. 우리가 현재 검증 중인 코드를 통해서 이를 확인해보도록 하자.

```
#include <string.h>
#include "MySUT.h"

INFO myInfo;

bool change_information(INFO *info) { /* 현재 단위 테스트 진행 중인 함수다 */
    INFO *oldinfo;

    if (info == NULL) {
        return false;
    }

    /* get_information()함수는 구현되지 않았지만, Mock에서 제공된다. */
    if ((oldinfo = get_information()) == NULL) {
        return false;
    }
    /* print_information()함수는 구현되지 않았지만, Mock에서 제공된다. */
    print_information(oldinfo);

    /* set_information()함수는 구현되지 않았지만, Mock에서 제공된다. */
    if (set_information(info) == NULL) {
        return false;
    }
    return true;
}
```

위에서 보듯이, 외부에 의존적인 첫 번째 코드는 “get_information()”이다. “get_information()”함수는 어떤 값을 돌려주어야 하며, 그 형식은 “INFO”의 포인터 이어야 한다. 따라서, 우리가 테스트 케이스에서 예측해야 할 값은 다음과 같이 코드로 표현될 것이다.

```
TEST( CMockTestGroup, changeInformationTestSuccess ) {
    INFO newInfo;
```

```

newInfo.name = "MJ Yoon";
newInfo.age = 44;
newInfo.alive= true;

get_information_ExpectAndReturn( &myInfo ); /* 미리 예측된 리턴 값을 설정해 준다. */

TEST_ASSERT_EQUAL_INT( true, change_information( &newInfo ) );
}

```

위의 코드에서 보듯이 목(Mock)에서 자동 생성된 "get_information_ExpectAndReturn()"을 이용해서, "get_information()" 함수가 호출되었을 때 돌려주어야 할 값을 설정해 주었다. 이제 단위 테스트를 다시 컴파일해서 실행하면 다음과 같은 결과가 보일 것이다.

Unity test run 1 of 1

```
... / s r c / T e s t / C M o c k T e s t C a s e . c : 2 9 : T E S T ( C M o c k T e s t G r o u p ,
changeInformationTestSuccess):FAIL:Function print_information. Called more times than
expected.
```

1 Tests 1 Failures 0 Ignored

FAIL

이전의 결과와 달라졌음을 볼 수 있다. 이번에는 "print_information()"이라는 함수가 호출되었지만, 우리는 그것이 호출되도록 만들어주지 않았다. 이제는 그 부분을 수정해서 다음과 같이 테스트를 변경하도록 한다.

```

TEST( CMockTestGroup, changeInformationTestSuccess ) {
    INFO newInfo;

    newInfo.name = "MJ Yoon";
    newInfo.age = 44;
    newInfo.alive= true;

    get_information_ExpectAndReturn( &myInfo );
    print_information_Expect( &myInfo ); /* 예측 값을 설정해 준다. */

    TEST_ASSERT_EQUAL_INT( true, change_information( &newInfo ) );
}

```

위와 같이 "print_information_Expect()"를 추가해 주었다. 즉, "print_information()" 함수가 호출될 수 있도록 목(Mock)을 설정해 준 것이다. 이때 중요한 것은 "print_information()" 함수는 복귀 값이 없다는 점이다. 목(CMock)에서는 복귀값이 없는 경우에는 "...Expect()"와 같이 목(Mock) 함수가 생성된다. 컴파일 후 실행하면 아래와 같은 결과가 보일 것이다.

Unity test run 1 of 1

```
... / s r c / T e s t / C M o c k T e s t C a s e . c : 2 9 : T E S T ( C M o c k T e s t G r o u p ,
changeInformationTestSuccess):FAIL:Function set_information. Called more times than
expected.
```

1 Tests 1 Failures 0 Ignored

FAIL

아직까지 테스트는 실패로 남아 있다. 이유는 아직 "set_information()"이 제대로 호출되지 않았기 때문이다. 이제는 마지막으로 "set_information()"함수가 호출될 수 있도록 목(Mock)을 설정할 차례다. 여기서 기존의 예와 다른 부분은 넘겨주는 파라미터와 복귀값이 둘 다 존재한다는 점이다. 아래와 같이 고치도록 했다.

```
TEST( CMockTestGroup, changeInformationTestSuccess ) {
    INFO newInfo;

    newInfo.name = "MJ Yoon";
    newInfo.age = 44;
    newInfo.alive= true;

    get_information_ExpectAndReturn( &myInfo );
    print_information_Expect( &myInfo );
    set_information_ExpectAndReturn( &newInfo, &myInfo ); /* 결과 리턴 값을 가진다. */

    TEST_ASSERT_EQUAL_INT( true, change_information( &newInfo ) );
}
```

위의 코드와 같이 "set_information()"함수를 위해서는 예측되는 값과 복귀 값을 둘 다 정해주었다. 각각을 "newInfo"와 "myInfo" 구조체의 주소로 정해주었다. 이와 같이 수정한 후에 실행하면 다음과 같은 결과를 볼 수 있을 것이다.

Unity test run 1 of 1

1 Tests 0 Failures 0 Ignored
OK

결과에서 볼 수 있듯이, 이제는 "changeInformationTestSuccess"라는 테스트 케이스가 통과 했음을 알 수 있다. 즉, "TEST_ASSERT_EQUAL_INT()" 매크로까지도 제대로 되었다는 것을 의미하기에, "change_information()" 함수가 "true"값을 돌려주었다고 생각할 수 있다. "change_information()" 함수를 보면, 전체 코드가 "get_information()", "print_information()", "set_information()"함수들에 의존적이지만, 그런 함수들이 없이도(구현하지 않고도) 단위 테스트를 할 수 있단 것을 알게 되었을 것이다.

6. 새로운 단위 테스트 케이스 추가하기

이제는 새로운 단위 테스트를 추가해 보도록 하자. 이번에는 실패하는 경우에도 제대로 동작 하는지 확인하기 위한 것이다. 만약, "set_information()"함수가 "NULL"값을 돌려 주었을 때, "change_information()"함수는 "false"를 돌려주어야 한다. 따라서, 테스트 케이스는 아래와 같이 작성할 수 있을 것이다.

```
TEST( CMockTestGroup, changeInformationTestFailure ) {
    INFO newInfo;

    newInfo.name = "MJ Yoon";
    newInfo.age = 44;
    newInfo.alive= true;
```

```

get_information_ExpectAndReturn( &myInfo );
print_information_Expect( &myInfo );
set_information_ExpectAndReturn( &newInfo, NULL ); /* 복귀 값을 변경했다. */

TEST_ASSERT_EQUAL_INT( false, change_information( &newInfo ) );
}

TEST_GROUP_RUNNER( CMockTestGroup ) {
    RUN_TEST_CASE( CMockTestGroup, changeInformationTestSuccess );
    /* 추가된 테스트 케이스 */
    RUN_TEST_CASE( CMockTestGroup, changeInformationTestFailure );
}

```

위의 코드에서 보듯이 테스트를 추가하기 위해서는 "RUN_TEST_CASE()"에 테스트 케이스를 추가해야 한다. 추가된 테스트 케이스에서는 "set_information()"이 "NULL"값을 돌려주도록 설정 했으며, "change_information()"이 그 값에 반응해서 "false"를 돌려주는지 확인했다. 즉, 단위 테스트를 하고 있는 함수가 외부의 입력에 의존성이 있는 경우 어떻게 테스트 하는지 확인한 것이다.

7. 같은 함수가 여러번 호출할 필요가 있는 경우에 목(Mock) 활용하기

만약, 의존성을 가지는 함수를 단위 테스트를 진행중인 함수가 여러번 호출할 경우에는 어떻게 할 것인가? 그 때는 앞에서 본 목(Mock) 함수를 여러 번 다른 값을 가지고 호출하는 방법을 취할 수 있다. 예를 들어, "while()" 루프와 같은 것을 돌면서 특정 값이 나오지 않는 때 계속 호출하는 경우를 대응해서 사용할 수 있다. 여기서는 두 번의 호출에 대해서 어떻게 대응 하는지 예제를 수정해 보도록 하겠다.

```

#include <string.h>
#include "MySUT.h"

INFO myInfo;

bool change_information(INFO *info) {
    INFO *oldinfo;

    if (info == NULL) {
        return false;
    }

    if ((oldinfo = get_information()) == NULL) {
        return false;
    }
    print_information(oldinfo);

    /* 여러번 의존하고 있는 함수를 호출하도록 한다. */
    while (set_information(info) == NULL) {
        /* Nothing to do here!!!*/
    }
    return true;
}

```

여러번 실행되는 것을 보기 위해서, "change_information()"함수를 조금 수정했다. 즉, "while()"루프를 이용해서 "set_information()" 함수가 "NULL" 값을 돌려주지 않을 때까지 실행했다. 이를 테스트 하

기 위한 테스트 케이스는 아래와 같다.

```

TEST( CMockTestGroup, changeInformationTestFailureMoreThanSuccess ) {
    INFO newInfo;

    newInfo.name = "MJ Yoon";
    newInfo.age = 44;
    newInfo.alive= true;

    get_information_ExpectAndReturn( &myInfo );
    print_information_Expect( &myInfo );
    set_information_ExpectAndReturn( &newInfo, NULL ); /* 여러번 NULL을 돌려준다. */
    set_information_ExpectAndReturn( &newInfo, NULL );
    set_information_ExpectAndReturn( &newInfo, NULL );
    set_information_ExpectAndReturn( &newInfo, &myInfo ); /* While()을 빠져 나온다. */

    TEST_ASSERT_EQUAL_INT( true, change_information( &newInfo ) );
}

TEST_GROUP_RUNNER( CMockTestGroup ) {
    RUN_TEST_CASE( CMockTestGroup, changeInformationTestSuccess );
    RUN_TEST_CASE( CMockTestGroup, changeInformationTestFailure );
    /* 테스트 케이스를 추가했다. */
    RUN_TEST_CASE( CMockTestGroup,
                   changeInformationTestFailureMoreThanSuccess );
}

```

테스트 케이스 "changeInformationTestFailureMoreThanSuccess"는 여러 번 "set_information()"의 호출이 발생하기에, 적당수 만큼의 예상값을 적어주었다 ("set_information_ExpectAndReturn()"). 물론, 각각의 경우에 "set_information()" 함수 호출의 인자를 다른 값으로 변경해도 상관없지만, 복귀 값은 "NULL"로 두어야 한다. 그래야만 단위 테스트 중인 코드의 "while()" 루프가 실행되는지 확인할 수 있다.

실행한 경과는 아래와 같다. 한 가지 주의할 점은 3개의 테스트 케이스 중에서 이전에는 성공한 케이스이지만, 이번에는 실패한 케이스가 만들어 졌다는 점이다. 이는 코드를 개발하는 과정에 흔히 발생할 수 있는 상황이기에 테스트 케이스가 타당한지 검토한 후에 코드를 수정하도록 하면 된다.

Unity test run 1 of 1

```
.... /src/Test/CMockTestCase.c:44: TEST(CMockTestGroup,
changeInformationTestFailure):FAIL:Function set_information. Called more times than
expected.
```

3 Tests 1 Failures 0 Ignored
FAIL

위의 결과에서 보듯이, "while()"루프를 추가하는 바람에 두 번째 테스트 케이스가 깨졌다. 따라서, 이때는 코드가 맞다고 판단한다면, 테스트 케이스를 변화된 코드에 맞게 수정해주는 것이 필요하다.

```
TEST( CMockTestGroup, changeInformationTestFailure ) {
    INFO newInfo;
```

```

newInfo.name = "MJ Yoon";
newInfo.age = 44;
newInfo.alive= true;

get_information_ExpectAndReturn( &myInfo );
print_information_Expect( &myInfo );
set_information_ExpectAndReturn( &newInfo, NULL );
set_information_ExpectAndReturn( &newInfo, &myInfo );

TEST_ASSERT_EQUAL_INT( true, change_information( &newInfo ) );
}

```

실패한 두 번째 테스트 케이스에 새로운 "set_information()" 함수의 예측값을 주었다. 이것을 통해서 제대로 수행된다면 "while()" 루프를 마치고 "true" 값을 돌려줄 것이라는 것을 알 수 있다. 실행의 결과는 아래와 같다.

Unity test run 1 of 1

...

3 Tests 0 Failures 0 Ignored
OK

[CppUTest를 이용한 단위 테스트]

앞에서 이미 “Unity”를 이용한 C언어 단위 테스트에 대해서 이야기 했다. “CppUTest”는 C/C++언어에서 사용할 수 있는 단위 테스트용 프레임워크로 “Unity”와 유사한 기능을 제공하지만, 목(Mock)도 포함해서 쓰기 간편하게 만들어진 프로그램이다. “CppUTest”는 C++로 작성된 코드이기에, C언어의 단위 테스트를 위해서 사용하기 위해서 헤더 파일들을 수정할 필요가 있을지도 모른다. 예를 들어, “extern "C" {}”와 같은 것을 이용해서 단위 테스트 대상이 되는 함수나 목(Mock)을 위한 헤더 파일들이 정의되어야 한다.

“Unity”와 비교해서 기능상의 차이는 없으며, 테스트를 만드는 방법도 거의 비슷하다. “CppUTest”로 작성된 테스트 케이스를 도구를 이용해서 “Unity”용으로 변경하는 것도 가능하다. “CppUTest”는 다양한 플랫폼을 대상으로 만들어져 있기에, 사용하기 전에 먼저 컴파일 과정을 거쳐서 라이브러리를 만들어 낸다. 만들어진 라이브러리와 헤더들은 특정 디렉토리에 설치가 가능하며, 나중에 컴파일하기 위해서 설치된 디렉토리를 접근하게 된다. “Unity”에서는 테스트 케이스들을 실행하기 위해서 그룹으로 묶고, 그룹의 특정 테스트 케이스를 일일이 실행되도록 처리해야 하지만, “CppUTest”에서는 그런 과정이 없다. 이것은 C++의 생성자(Constructor)를 이용해서 자동으로 수행된다.

1. “CppUTest”的 설치

“Windows”와 같은 환경에서는 “Visual Studio”를 위한 설치 방법도 있지만, 여기서는 “Windows” 환경의 “Eclipse”와 “MinGW”라는 컴파일러를 이용하는 방법에 대해서 설명 하겠다. “Mac OS X”에서는 “bash” 쉘을 실행해서 커맨드로 입력해도 마찬가지로 설치할 수 있다. 설치를 위해서 필요한 프로그램들은 아래와 같다.

1. Eclipse CDT : <https://www.eclipse.org/downloads/>
2. MinGW : <https://sourceforge.net/projects/mingw/files/>
3. CppUTest : <https://cpputest.github.io/>

차례로 선택해서 다운로드 받아서 설치하면 될 것이다. “CppUTest”的 경우에는 윈도우에서 다운받아 다시 컴파일을 거쳐야 하기 때문에 바로 설치되지 않는다. 그냥 소스 코드 형태로 받을 수 있을 것이다. “Windows”의 커맨드 창을 열어서 “MinGW”에서 설치한 명령어들을 사용할 수 있는지 확인하는 절차도 필요하다. 만약, 명령어가 실행되지 않는다면 환경 변수 설정이 문제가 있기에, 환경 변수 중 “Path”的 설정을 “MinGW”가 설치된 디렉토리의 “bin” 디렉토리를 접근할 수 있도록 추가해 주어야 한다.

이제는 커맨드 창이 열린 상태에서 설정 및 컴파일, 설치를 진행할 차례다. 일반적으로 “Unix”와 같은 계열의 운영체제를 사용하는 시스템을 위해서 소스코드 형태로 배포되는 소프트웨어는 자동으로 컴파일을 설정할 수 있는 기능을 제공한다. 아래와 같은 명령어를 입력해서 진행하도록 하자.

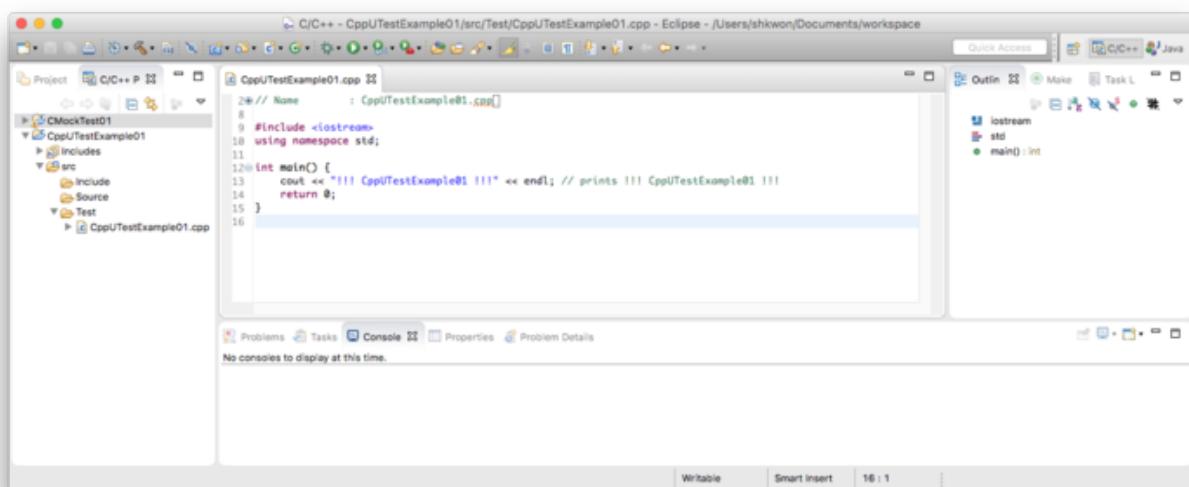
1. > sh ./configure
2. > make
3. > make install

“Windows”개발 환경에서는 설치한 “MinGW”에서 명령행을 이용해서 위의 명령들을 실행하면 될 것이다.

설치가 제대로 되었다면, 기본적으로 설치 되는 다음의 디렉토리에서 관련 파일들을 찾을 수 있을 것이다. “/usr/local/lib”에 대한 “MinGW”는 “C:\MinGW\msys\1.0\local”이하의 디렉토리를 살펴보면 된다. 각종 헤더 파일들을 포함한 “include”와 라이브러리로 “libCppUTest.a”와 “libCppUTestExt.a” 파일들을 찾을 수 있어야 한다.

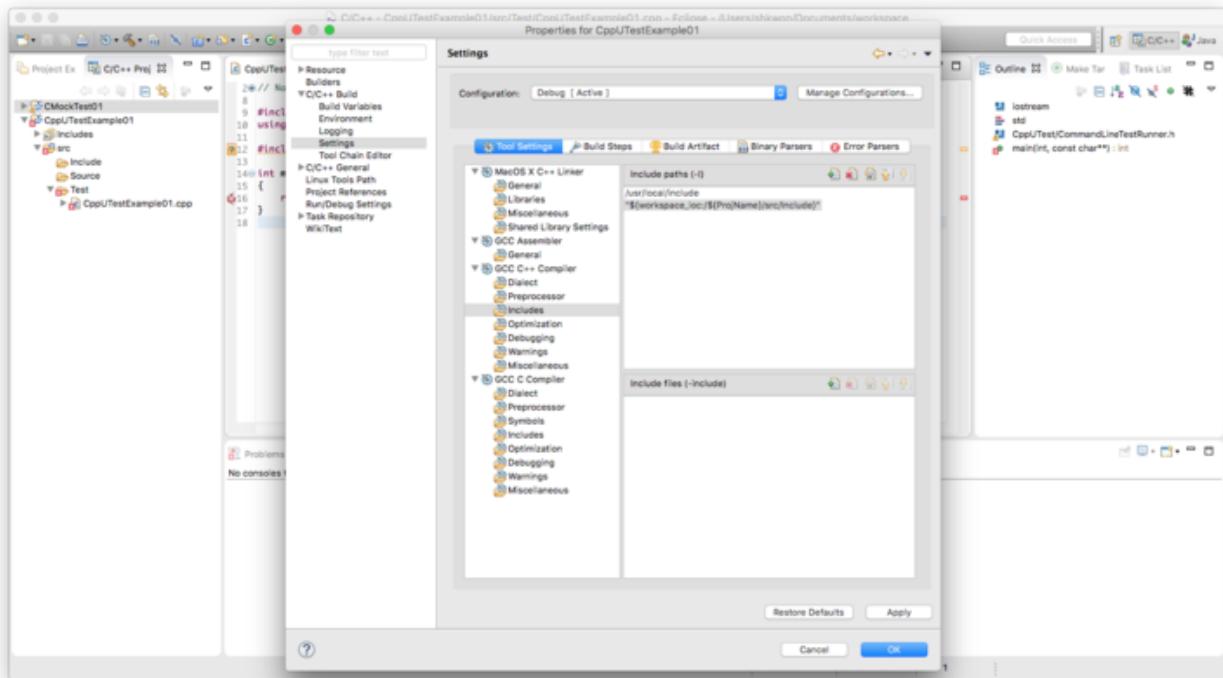
2. Eclipse 프로젝트의 생성

C언어의 함수를 단위 테스트하지만, 기본적으로 C++로 단위 테스트를 만들어야 하기에, “Eclipse”的 프로젝트는 C++로 생성하도록 한다. 프로젝트를 생성한 후에는 “CppUTest” 라이브러리 들을 이용하기 위해서(프로젝트에 포함하기 위해서) “Project->Properties” 설정을 변경해 주어야 한다. 물론, 라이브러리는 최종적으로 실행 이미지를 만들때 링커(Linker)에 의해서 사용 되기에, 사용하는 링커의 옵션에 반영해 주어야 한다. 추가적으로 한 가지 주의할 점은 C파일과 C++파일들에 설정을 각각 명시해 주어야 한다는 점이다. 즉, C에서 필요한 “include” 디렉토리와 C++에서 사용하는 “include” 디렉토리의 설정을 각각 필요에 맞게 추가해 주어야 한다.

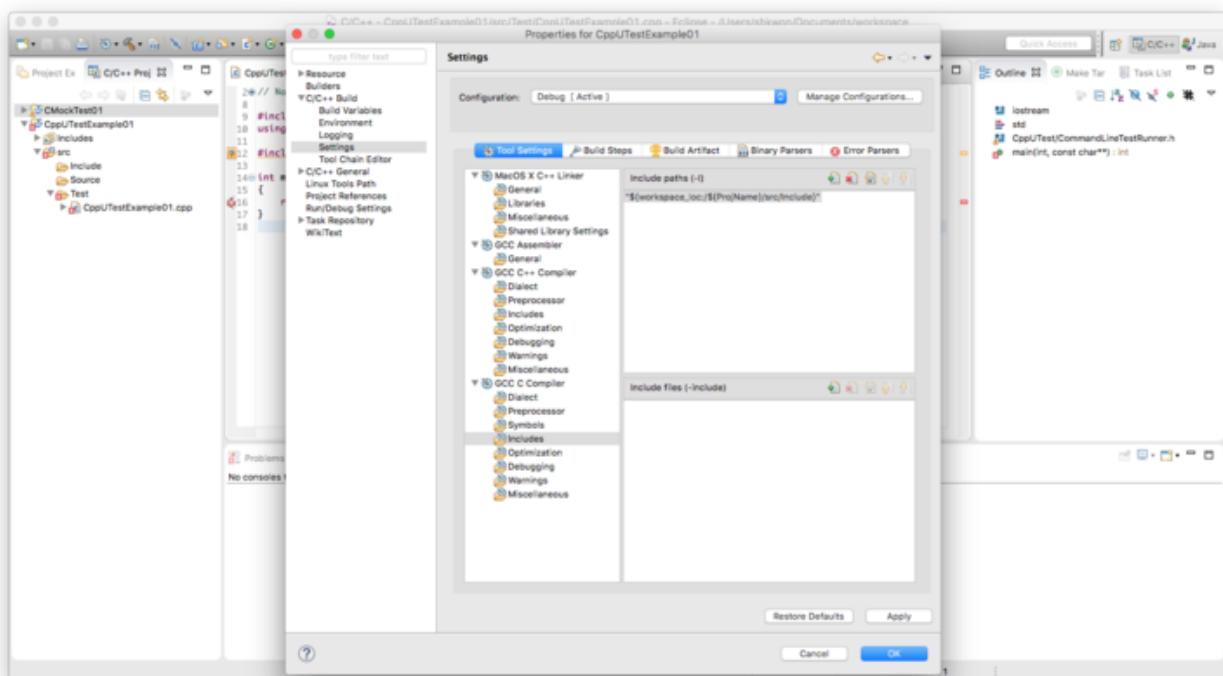


그림에서 보듯이 디렉토리의 구조는 테스트 케이들을 위한 “Test”, 단위 테스트를 하고 있는 코드를 위한 “Source”, 헤더 파일들을 저장할 “Include”를 만들었다. “Test” 디렉토리 이하에는 테스트 실행을 위한 “main()”함수를 가지는 파일을 넣었다. “main()”함수에는 “CppUTest”的 코드를 시작하기 위해서

는 아래와 같이 코드를 넣어야 한다.



그림과 같이 "C/C++ Compiler"를 위한 "include" 디렉토리에는 "CppUTest"의 헤더를 찾을 수 있는 "/usr/local/lib/include"가 포함되어야 하며, 단위 테스트를 위해서 프로젝트의 "workspace"에 있는 "include" 디렉토리도 포함시켜야 한다.



"C Compiler"를 위해서는 그림과 같이 프로젝트의 "workspace"에 있는 "include" 디렉토리만 추가하면 된다. 이제는 기본적인 설정이 끝났기에 단위 테스트를 위해서 코드를 작성해 보도록 하겠다. 가장

먼저 “main()” 함수 부터 보도록 하자.

3. 단위 테스트의 main() 함수 작성

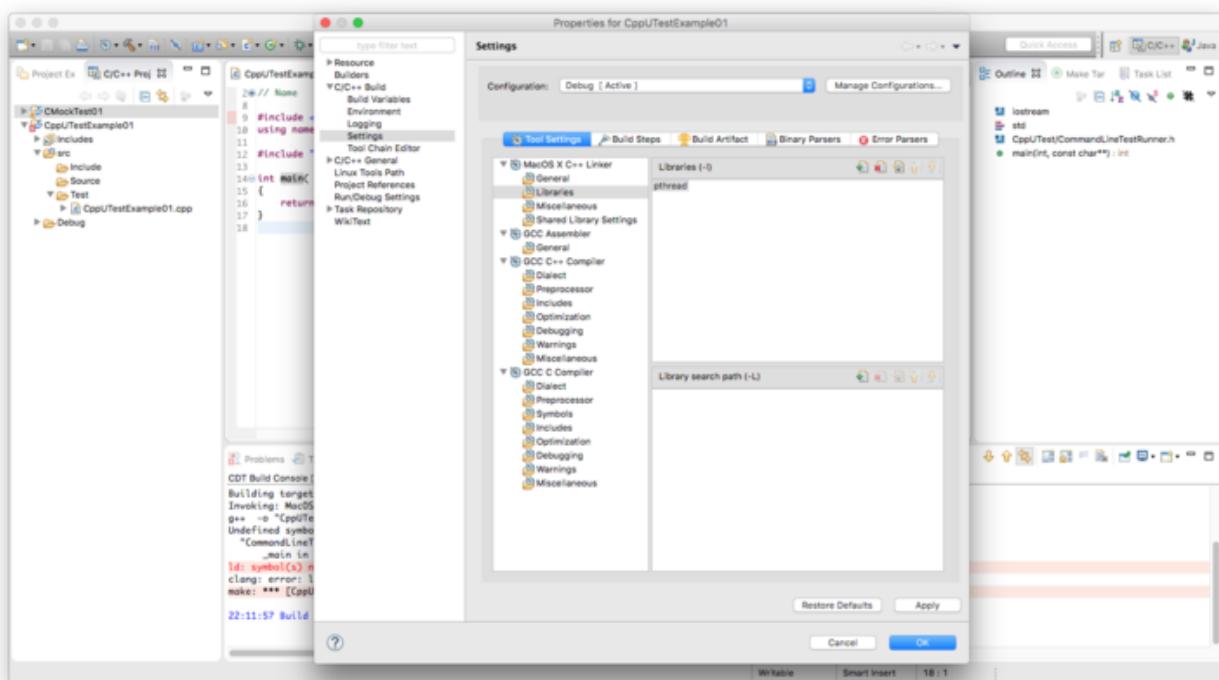
“main()” 함수에서는 “Unity”와 마찬가지로 테스트 그룹(Group)을 실행하도록 해주는 부분이 들어간다. 테스트 실행기(Test Runner)의 “RunAllTests()” 메서드(Method)를 호출하는 것으로 시작한다. 코드는 아래와 같다.

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

#include "CppUTest/CommandLineTestRunner.h"

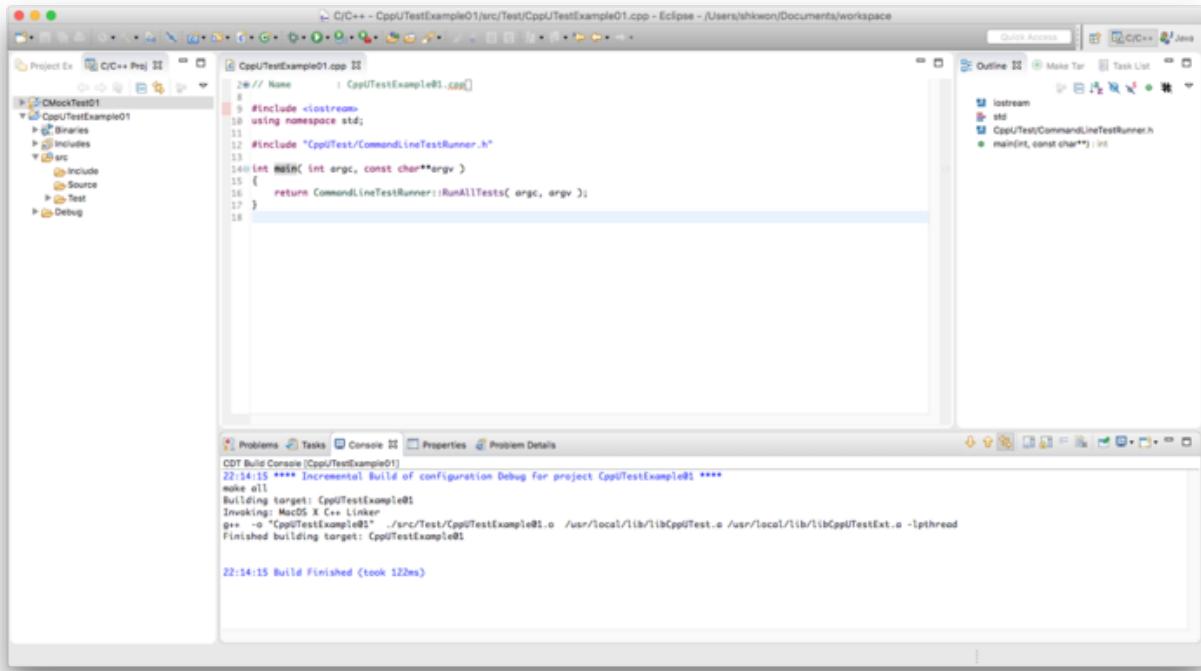
int main( int argc, const char**argv ) {
    return CommandLineTestRunner::RunAllTests( argc, argv );
}
```

이 코드에서 “#include <iostream> ... using namespace std;” 부분은 기본적으로 “Eclipse”에서 자동으로 생성해 주는 부분으로 변경없이 사용했다. 하지만, 나머지 코드들은 직접 입력해야 할 것이다. 이 상태에서 그냥 컴파일 시키면, “CppUTest”에서 필요한 라이브러리를 찾을 수 없기에 실행 이미지가 생성되지 않는다. 따라서, 다음 그림과 같이 “CppUTest”와 “Pthread(Posix Thread)” 라이브러리를 프로젝트에 포함시켜 주어야 한다. “Pthread” 라이브러리는 테스트 케이스를 실행하기 위해서 필요한 쓰레드(Thread)를 만들기 위해서 사용되는 “POSIX” 쓰레드 관련 함수 들의 모음이다.



한 가지 주의할 점은 “MinGW”와 같은 링커(Linker)를 사용한다면, “pthread”를 “-l” 옵션에 추가해 주어야 하며, “CppUTest”와 “CppUTestExt”는 “Miscellaneous-->Other objects”에 추가해야 한다. 파일의 이름을 전체 경로에서 찾아서 넣어주면 된다. 이제 기본적인 컴파일이 가능한 상태가 되었기에, 프로젝트의 빌드(Build)를 실행해 볼 수 있다. “MinGW”的 경우에는 컴파일시에 “경고(Warning)(#define XXX가 중복되었다는)” 메시지가 발생할 수 있는데, 이때는 설치된 “CppUTest”

의 헤더 파일을 일부 수정(주석으로 처리)해도 된다.



아직은 아무런 테스트도 작성한 것이 없기에, 아래와 같이 단순한 출력만 보여줄 것이다. 이제부터는 해야 할 일은 간단하게 “CppUTest”에서 어떻게 목(Mock)을 사용하는지 보도록 할 것이다.

OK (0 tests, 0 ran, 0 checks, 0 ignored, 0 filtered out, 0 ms)

[CppUMock을 이용한 단위 테스트]

“Unity”와 마찬가지로 “CppUTest”도 자신만의 목(Mock)을 가지고 있다. “Unity”와 다른 점은 “CppUTest”는 자체적으로 “CppUMock”이라는 목(Mock)을 같이 가지고 있으며, 별도의 라이브러리로 생성되어 설치된다(“CppUTestExt.a”). 여기서 보여주는 예제는 이미 “CMock”에서 보았던 것을 “CppUTest”的 목(Mock) 형식으로 바꾼 것이다.

1. Header파일들

“CppUMock”을 만들기 위해서는 목(Mock)으로 생성(Mock으로 대체)될 함수 들에 대한 기본 선언을 필요로 한다. 즉, 어떤 자료구조를 사용하며, 어떤 인수와 복귀 값을 가지는지 미리 정의해 주어야 한다. 특히, 헤더 파일인 경우에는 C가 아닌 C++에서 사용될 것이라는 것을 알고 작성해야 한다.

```
/* Include_informationhandler.h */
#ifndef INCLUDE_INFORMATIONHANDLER_H_
#define INCLUDE_INFORMATIONHANDLER_H_

#ifndef __cplusplus /* C로 작성된 함수를 CppUTest(C++)로 단위 테스트 할 것이다. */
extern "C" {
#endif

#include <stdbool.h>

typedef struct INFORMATION {
    char *name;
```

```

int age;
bool alive;
} } INFO;

INFO* get_information( void ); /* 구현되지 않은 Mock으로 대체될 함수들 */
void print_information( INFO *info );
INFO* set_information( INFO *info );

#ifndef __cplusplus
}
#endif
#endif /* INCLUDE_INFORMATIONHANDLER_H_ */

```

위의 코드에서 보듯이 “Unity”와는 달리 “#ifndef __cplusplus ~ #endif”가 포함된 것을 볼 수 있을 것이다. 이것은 C로 정의된 함수를 C++에서 사용할 수 있도록 만들기 위해서 추가되었다. “CppUTest”는 원래 C/C++의 단위 테스트 용으로 만들어졌기 때문이다.

```

/* include_changeinformation.h */
#ifndef INCLUDE_CHANGEINFORMATION_H_
#define INCLUDE_CHANGEINFORMATION_H_

#ifndef __cplusplus
extern "C" {
#endif

extern INFO myinfo; /* 단위 테스트 대상이 되는 함수 */
bool change_information( INFO *info );

#ifndef __cplusplus
}
#endif
#endif /* INCLUDE_CHANGEINFORMATION_H_ */

```

위의 코드는 단위 테스트의 대상이 되는 함수를 정의하고 있다(“change_information()”). 전역변수로 사용할 “myinfo”는 extern으로 선언되어, 나중에 C구현 파일로 들어갈 것이다. 이 헤더 파일도 마찬가지로 “CppUTest”와 같이 컴파일하기 위해서 “#ifndef __cplusplus ~ #endif”를 사용해서 정의되었다.

2. 단위 테스트 대상 함수의 정의

이제는 단위 테스트의 대상이 될 코드(SUT)보도록 하자. 여기서는 아직 구현되지 않은 함수들을 나중에 목(Mock)으로 대체해 준다는 것을 가정했다. 하지만, 구현은 되지 않았을지 모르지만 기본적으로 어떤 일을 처리하고 어떤 결과를 돌려줄 것인지 정도는 미리 정의하고 있어야 한다. 즉, 사용 되는 각 함수의 인터페이스(Interface) 정의는 가지고 있다고 가정한다.

```

/* change_information.c */
#include <stdbool.h>
#include <string.h>

#include "InformationHandler.h"
#include "ChangeInformation.h"

INFO myinfo;

```

```

bool change_information( INFO *info ) {
    INFO *oldinfo;

    if( info == NULL )
    {
        return false;
    }

    oldinfo = get_information( );
    if ( oldinfo == NULL )
    {
        return false;
    }

    print_information( oldinfo );

    if( set_information( info ) == NULL )
    {
        return false;
    }
    return true;
}

```

위의 코드는 단위 테스트 대상이 되는 함수를 정의하고 있다. 이 파일은 순수 C언어로 작성된 파일이다. 나중에 “Eclipse”에서 컴파일 시에 “C Compiler(GCC 또는 MinGW)”를 이용하게 될 것이다. 추가적으로 한 가지 이야기 할 것은 외부에서 접근할 수 있는 전역변수로 “myinfo”의 정의 자체는 C구현 파일에 위치시키고 있다.

3. 테스트 케이스 구현

테스트 케이스는 크게 테스트 그룹의 선언과 테스트 케이스로 구분되며, “Unity”와는 다르게 목(Mock)을 자동으로 생성해 주지 않고 수동으로 만들어야 한다. 각각이 장단점이 있기에 어떤 것이 더 좋다고 이야기 하기는 힘들다. 처음 사용하는 사람들은 어떻게 사용하는지 난감해 할 수도 있으며, 작은 입력 오류를 찾는 것도 쉽지 않은 편이다. 따라서, 직접 손으로 입력해서 오류가 어떤 상황에 발생하는지를 눈여겨 볼 필요가 있다.

```

/* cpputestcase.c */
#include "CppUTest/TestHarness.h"
#include "CppUTestExt/MockSupport.h"

#include "InformationHandler.h"
#include "ChangeInformation.h"

```

“CppUTest”的 목(Mock)을 사용하기 위해서는 “#include “CppUTestExt/MockSupport.h””가 필요하다. 그리고, 테스트 케이스를 위해서 “CppUTest/TestHarness.h”도 포함되어야 한다. 나머지 두 개의 헤더는 앞에서 보았던 목(Mock)으로 대체할 헤더 파일과 단위 테스트의 대상이 되는 함수의 정의를 가지고 있는 헤더 파일이다.

```

// To delete warning message in pointer initialization to constant string
#pragma GCC diagnostic ignored "-Wwrite-strings" // Turn-off compiler warning option.

```

C++에서 경고(Warning) 메시지를 없애기 위해서 위와 같이 “constant string” 을 "char*"에 지정하는 경고를 없애주도록 만들었다. 따라서, 이 부분은 반드시 필요한 것은 아니다.

```
INFO newinfo;
```

```
// Define Mock Type Comparator
class MyTypeComparator: public MockNamedValueComparator {
public:
    virtual bool isEqual(const void *object1, const void *object2)
    {
        return object1 == object2;
    }
    virtual SimpleString valueToString(const void *object)
    {
        return StringFrom(object);
    }
};
```

테스트에서만 사용될 전역변수 "newinfo"를 선언했다. 이것은 편의에 따른 것으로 필요한 부분에 선언하면 된다. 비교자(Comparator)는 목(Mock)이 기대된 타입의 값을 제대로 주고 받았는지를 비교하기 위해서 사용하는 클래스다. 이곳에 정의된 것을 예로, 자신에게 필요한 비교자를 만들면 된다. 여기서는 나중에 "INFO" 타입을 비교하기 위해서 목(Mock)에서 사용 되지만, 직접적으로 비교하는 것은 자동으로 이루어지는 과정이라 코드는 따로 없다.

```
// Define Test Group here!!!
TEST_GROUP( CHANGE_INFORMATION ) {
    MyTypeComparator comparator; /* 비교자 생성 */
    // Setup()
    void setup() {
        // Install Type Comparator
        mock().installComparator("INFO*", comparator);
        // Global information
        myinfo.name = "Suho Kwon";
        myinfo.age = 44;
        myinfo.alive = true;

        // New information
        newinfo.name = "MJ Yoon";
        newinfo.age = 44;
        newinfo.alive = true;
    }
    // Teardown()
    void teardown() {
        // Restore default value
        newinfo.name = "";
        newinfo.age = 0;
        newinfo.alive = false;

        // Uninstall Type Comparator
        mock().removeAllComparators();
        // Clear Mock
    }
}
```

```

        mock().clear();
    }
};

```

“Unity”와 달리 “CppUTest”에서는 테스트 그룹을 마치 클래스 처럼 선언하며, 테스트 그룹의 이름도 명시했다(CHANGE_INFORMATION). 내부는 새로운 비교자를 하나 생성하고, “Setup”과 “Tear-down”함수를 정의했다. “Setup”에서는 비교자를 목(Mock)에 등록하기 위해서 “installComaparator()”를 호출 했으며, “Tear-down”에서는 그것을 다시 제거하기 위해서 “removeAllComparator()”를 호출했다. 추가적으로 “Tear-down”에서는 목(Mock)의 “clear()”를 호출해서 사용하고 있는 메모리 공간의 누수(Leak)를 방지해주었다. 항상 사용된 목(Mock)의 메모리 공간은 안정적으로 제거하는 것이 좋다.

```

// Define Mock
INFO* get_information( void ) {
    return
        (INFO*) mock().actualCall("get_information").returnPointerValueOrDefault(&myinfo);
}

void print_information( INFO *info ) {
    mock().actualCall("print_information").withParameterOfType("INFO*", "info",
&myinfo);
}

INFO* set_information(INFO *info) {
    return (INFO*) mock().actualCall("set_information").withParameterOfType("INFO*",
"info", &newinfo).returnPointerValueOrDefault(&myinfo);
}

```

앞에서 이야기 했듯이 “CppUTest”는 “Unity”와 달리 자동으로 목(Mock)을 생성하지 않고, 위와 같이 직접 코드를 작성해야 한다. 이 목(Mock)들을 자세히 보면 알 수 있듯이, “Return” 값의 유무, “Parameter”的 유무 등을 직접 표현해 주어야 한다. 여기서 사용 되는 함수의 원형은 앞에서 본 헤더 파일의 함수 원형 선언과 일치하며, “Return” 값이 있을 경우에는 “return”을 사용해서 실제 호출(“actualCall()”)의 결과 값을 돌려주어야 한다. “actualCall()”에서 넘겨받는 객체(Object)의 메쏘드(Method)들을 보기 위해서는, “Eclipse”에서 입력을 잠시 멈추면 관련된 다양한 메쏘드(Method)들을 자동으로 보여줄 것이다. 그 중에서 선택해서 자신의 “Return” 타입에 적합한 것을 선택해서 사용하면 된다.

```

// Test script starts here!!!
TEST( CHANGE_INFORMATION, ChangeInformationSuccess ) {
    // 함수 이름, 파라미터 타입, 파라미터 이름, 함수에 전달될 파라미터 값의 순서로 사용한다.
    // 예상된 호출을 미리 기록해 둔다. 나중에 테스트 실행시 호출 되었는지 확인할 것이다.
    mock().expectOneCall("get_information").andReturnValue(&myinfo);
    mock().expectOneCall("print_information").withParameterOfType("INFO*", "info",
&myinfo);
    mock().expectOneCall("set_information").withParameterOfType("INFO*", "info",
&newinfo);

    LONGS_EQUAL(true, change_information(&newinfo)); // 단위 테스트 실행

    // 호출을 비교하기 위해서 비교자를 선언해 주었다.
}

```

```

    mock().checkExpectations(); // 단위 테스트 실행 후 예상된 호출과 실제 호출을 비교한다.
}

```

테스트 케이스는 동일한 테스트 그룹에 속하며(CHANGE_INFORMATION), 각각은 이름을 가지고 있다. 각각의 테스트 케이스마다 적절한 이름을 부여하면 된다. 단위 테스트되고 있는 함수를 호출하기 전에, 그 함수가 의존하고 있는 함수들의 목(Mock)을 설정하기 위해서 "expectOneCall()"과 같은 목(Mock)의 메쏘드(Method)를 호출했다. 이 메쏘드(Method)도 다양하게 준비되어 있으니, 목적에 맞는 것을 찾아서 사용하면 된다. "Return" 값이 있는 경우에는 "andReturn()"을 이용해서 목(Mock)이 호출되었을 때 "Return"으로 돌려 받을 값을 명시한다. 함수가 그외의 다른 "Parameter"들을 가지고 있다면, 함수의 타입정보와 "Parameter"의 이름, 실제로 "Parameter"로 넘겨질 값을 명시한다.

"Unity"와 마찬가지로, 단위 테스트 함수를 호출한 후에, "Return" 값을 비교해서 제대로 동작했는지를 확인할 수 있다("LONGS_EQUAL()", 즉 호출 결과가 올바른지 확인). 이러한 매크로는 "Unity"와 거의 비슷하게 제공되고 있으니, 메뉴얼을 참고해서 사용하면 될 것이다. 목(Mock)에 대해서 기대값으로 설정된 모든 함수와 파라미터 및 "Return" 값들이 사용되었는지 확인하기 위해서 "checkExpectations()"를 호출했다. 만약, 바로 앞에서 했던 함수 호출 결과가 오류가 있었다면, 이 부분은 실행되지 않을 것이다. 만약 앞에서 확인했던 함수 호출 결과가 문제가 없더라도, 이곳에서의 검사가 만족되지 않으면 테스트는 실패로 간주된다.

```

// Test script starts here!!!
TEST( CHANGE_INFORMATION, ChangeInformationGetInformationFailure ) {
    mock().expectOneCall("get_information").andReturnValue((void*)NULL);
    //mock().expectOneCall("print_information").withParameterOfType("INFO*", "info",
&myinfo); // Mock 설정을 막았다.
    //mock().expectOneCall("set_information").withParameterOfType("INFO*", "info",
&newinfo); // Mock의 설정을 막았다.

    LONGS_EQUAL(false, change_information(&newinfo));
    mock().checkExpectations();
}

```

위의 코드는 앞의 테스트와 유사하지만, 이번에는 "get_information()"함수가 "NULL"값을 복귀값으로 돌려주는 경우를 테스트로 보여준다. 포인터 타입에 대한 문제로 컴파일 오류가 발생해서 "void *"으로 형변환을 해서 복귀값으로 주었다. "get_informatoin()"함수에서 "NULL"을 돌려줄 경우, "change_informatoin()"함수는 중간에 "false" 리턴 할 것이므로, 두 개의 남은 목(Mock) 기대값은 사용 되지 않을 것이다. 여기서는 그냥 단순히 코멘트로 처리해 주었다. 함수의 결과값도 당연히 "false"와 같은지 비교했다.

4. 테스트 결과의 확인

테스트 결과는 테스트 케이스의 실행 결과를 알려 준다. 이곳에서는 인위적으로 테스트 케이스를 실패시키기 위해서 코드를 수정 했다.

```

.../src/Test/ChangeInformationTest.cpp:94: error: Failure in
TEST(CHANGE_INFORMATION, ChangeInformationGetInformationFailure)
Mock Failure: Expected call did not happen.
EXPECTED calls that did NOT happen:
    print_information -> INFO* info: <0x100365f58>
    set_information -> INFO* info: <0x100365ea0>
ACTUAL calls that did happen (in call order):
    get_information -> no parameters

```

..
Errors (1 failures, 2 tests, 2 ran, 10 checks, 0 ignored, 0 filtered out, 1 ms)

위의 단위 테스트 실행 결과는 앞에서 코멘트로 처리된 두 개의 목(Mock) 기대값 설정을 돌려놓은 후에 나오는 오류 메시지다. "print_information()"과 "set_information()" 함수가 호출되지 않아서 오류라는 것을 보여준다. 만약, 코멘트로 만들어놓고 실행한다면 아래와 같은 결과를 볼 수 있을 것이다.

..
OK (2 tests, 2 ran, 8 checks, 0 ignored, 0 filtered out, 1 ms)

두 개의 테스트 케이스가 제대로 실행 되었다는 것을 알 수 있다. 체크된 회수는 목(Mock)에서 6번, 함수 리턴 값 비교가 2번이라 총 8회가 된다. 무시(Ignore)나 제외(filtered out)은 사용 되지 않았다. 더 자세한 사용법을 알고자 한다면, "CppUTest"에서 제공하는 메뉴얼을 참고해서 보면 될 것이다. 메뉴얼에서 이해하기 어려운 부분은 이곳에서 제공하는 실제 예제에서 유추하면 좀 더 쉽게 이해할 수 있을 것이다. "CppUMock"의 메뉴얼은 아래의 링크에 있다.

https://cpputest.github.io/mockng_manual.html

[테스트 범위(Test Coverage)의 측정]

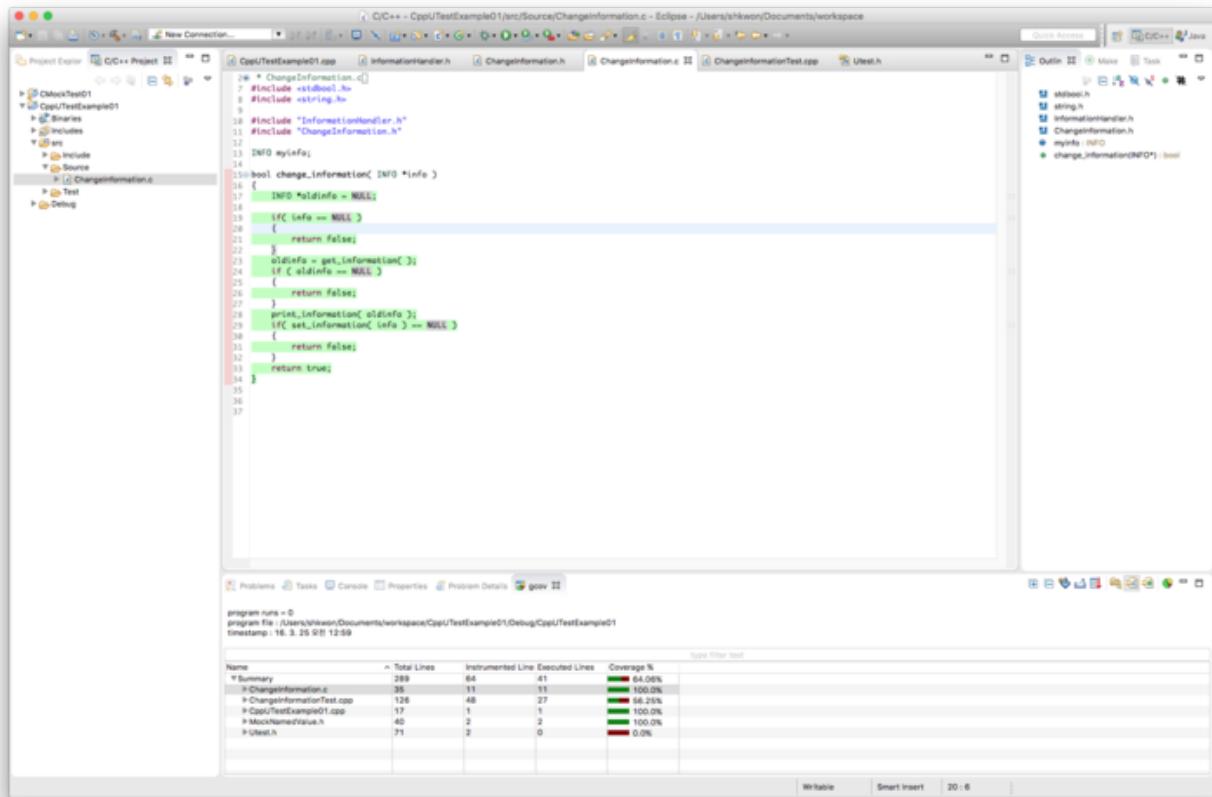
테스트가 되는 코드를 가졌다고 안심하기는 이르다. 우리가 만든 테스트가 실제로 얼마나 많은 코드를 실행하고 있는지 알아야 한다. 그렇게 하기 위해서는 테스트 커버리지(Test Coverage)를 측정하는 방법이 필요하다. 여기서는 Eclipse를 이용해서 어떻게 테스트 커버리지를 측정하는지 알아보도록 하겠다.

테스트 커버리지 정보가 있다면, 이 정보를 이용해서 테스트 되지 않는 코드의 테스트를 위해서 테스트 케이스를 보완하는데 사용할 수 있다. 테스트 커버리지 정보에는 함수, 코드 라인, 조건문에 대한 것들이 포함되며, 함수 커버리지가 가장 넓으며, 코드 라인은 그나마 쉽게 구할 수 있다. 어려운 부분은 조건문에 따른 분기를 어떻게 테스트 케이스에 반영할 것인지를 고민하는 것이다.

이와 같은 것을 볼 때, 함수는 될 수 있으면 한 가지 일만하고, 짧아야 하며, 파라미터가 없을수록, 조건문이 복잡하지 않아야 테스트가 하기 쉽다는 것을 알 수 있다. 복잡한 조건문은 테스트 케이스를 만들기 어렵게하고, 파라미터가 많으면, 조건으로 주어져야 할 입력이 늘어나게 되며, 코드가 길어지면 한가지 일을 하지 않고 복합적인 일을 하기에 테스트를 하기 위해서 의존적인 부분을 대체해야 할 일이 많이 생긴다. 따라서, 가능한 테스트하기 쉬운 함수를 만드는 것이 일을 수월하게 만든다. 추가적으로 모든 문장이 실행된다고 해서 버그가 없다고 생각해선 안된다. 테스트 케이스 입력 값에 따라 다른 결과가 나올 수 있기 때문이다.

Unix와 같은 시스템에서는 오래 전부터 이런 부분들에 대해서 다양한 작업이 진행되어 왔으며, 커버리지 측정을 위해서 오픈소스로 만들어진 것들이 있다. GCC와 같은 계열의 컴파일러를 사용한다면, "GCOV"(GNU Coverage)와 같은 툴들은 커버리지 측정을 위해서 사용되며, 이것을 웹 형식으로 보여주기 쉽도록 만들기 위해서, "LCOV"를 이용한다. "GENHTML"은 "HTML"로 커버리지 정보를 변환하기 위해서 사용하는 툴이다. 따라서, 결과를 보기 위해서 웹 브라우저를 사용할 수 있다.

각각의 프로그램에 대한 설치나 사용법은 이곳에서 이야기 하지 않도록 하겠다. 지금까지 작성한 테스트 예제를 이용해서, 테스트 케이스를 보완하는 것을 주로 이야기 할 것이다. 즉, 테스트 케이스가 커버하지 않는 코드를 테스트 케이스를 보완해서 실행되도록 만들며, 조건문의 분기와 같은 경우도 마찬가지로 참/거짓을 둘 다 실행하기 위한 테스트 케이스를 간단히 어떻게 구현 하는지 볼 것이다. 아래는 지금까지 작성한 테스트 케이스를 실행한 것을 보여준다.



위의 그림은 단위 테스트 중인 코드의 커버리지를 보여준다. 녹색으로 표시된 코드는 실행 되었다는 것을 나타낸다. 화면의 하단분에는 프로젝트에 관련된 모든 코드의 커버리지 정보를 알 수 있다. 이와 같은 결과를 보기 위해서는 프로젝트 빌드 설정에서 C 및 C++ 컴파일러의 설정("Project-->Properties-->Settings-->Tool Setting-->[C 및 C++ Build]-->Debugging")에서 "GCOV"의 정보를 생성하도록 체크 해야 한다("-fprofile-arcs -fprofile-gcov"). 물론, "GCOV Plugin"이 이미 설치되어 있어야 제대로 동작할 것이다.

"GCOV Plugin"은 Eclipse Plugin에서 찾아서 설치하면 된다. Linux Tool의 일부로 제공 되기에, "Help-->Install New Software"에서 찾아보기 바란다. 위와 같은 화면을 보기 위해서는, "Project"에서 생성되는 "Debug" 관련 폴더를 삭제한 후에 다시 컴파일할 필요가 있다. 만약, 이상한 데이터가 나온다고 생각되면, "Debug"을 지우고 다시 컴파일 하도록 한다. 위의 정보는 "Project-->Profiling Tools-->Profile Code Coverage"를 선택하면 된다. 미리 빌드(Build)한 후에 실행해야 하며, 그렇지 않은 경우에는 실행할 "Binary Image"를 찾지 못해서 오류 메시지를 보일 것이다.

커맨드 라인에서 생성된 프로파일 정보를 웹브라우저에서 보기 위해서는 "LCOV"와 "GENHTML"이 필요하다. 각각은 "GCOV"를 통해서 생성된 정보를 읽어서 "HTML" 형식의 파일로 만들어주며, 코드와 함께 웹 브라우저에서 커버리지 정보를 확인할 수 있다. 명령은 아래와 같다.

```
> lcov --rc lcov_branch_coverage=1 --capture --directory gcov_data_file_directory --output-file generated.info
> genhtml generated.info --branch-coverage --output-directory generated_html_directory
```

"gcov_data_file_directory"는 "GCOV"에서 생성한 정보가 저장되어 있는 디렉토리를 말하며, "generated.info"는 "GCOV"의 데이터를 변화시켜서 나온 정보 파일이다. 이때, 분기 커버리지를 보기 위해서는 옵션으로 "--rc lcov_branch_coverage=1"을 주어야 한다. 마찬가지로 "genhtml" 명령어에

도 분기 커버리지를 보기 위해서 "--branch-coverage"를 옵션으로 주었으며, 생성되는 "HTML"파일들의 저장을 위해서 "generated_html_directory"를 명시했다.

[참고]

예를 들어, Mac OS X El Capitan에서 Eclipse를 사용하고 있다면, 아래의 링커(Linker)옵션을 설정 해주지 않으면, "GCOV" 옵션으로 컴파일이 안될 것이다. 반드시 링커(Linker) 옵션으로 설정하기 바란다. 아래에서 "7.0.2"는 Mac OS X El Capitan버전의 경우에 해당하며, 만약 다른 버전의 Mac OS X를 사용한다면, 그 번호를 적절히 변경해 주면 된다. 아래의 옵션에서 "7.0.2"와 같은 부분은 자신의 컴퓨터에 설치된 "XCode"의 버전에 따라 다르다. 따라서, 직접 확인하고 사용하기 바란다. 현재 (2016.03.26) 기준으로 업데이트 된 XCode버전은 "7.3.0"이다.

```
-Iclang_rt.profile_osx
-L/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/
lib/clang/7.0.2/lib/darwin
```

Mac OS X의 경우 "GCOV" 커맨드 라인 툴은 기본적으로 설치되어 있지만, "LCOV"와 "GENHTML"은 설치 해주어야 한다. 커맨드 라인에서 다음과 같이 하도록 한다. 위의 내용을 프로젝트 설정에서 변경하기 위해서는 "Project->Properties->[사용하는 컴파일러의 링커]->Libraries"에서, "-I"에는 "clang_rt.profile_osx"를 적어주어야 하며, "-L"에는 "/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/lib/clang/[사용하는 XCode의 버전번호]/lib/darwin"를 적어주어야 한다. 이를 확인하기 위해서는 명령어 라인(Command Line)에서 "Unix"명령어를 이용해서 해당하는 디렉토리로 가보면 "XCode"의 버전번호를 확인할 수 있을 것이다.

```
> ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
</dev/null 2> /dev/null [enter]
brew install lcov [enter]
```

[문제점]

만약, Mac OS X El Capitan에서 Eclipse를 사용하고 있다면, 커버리지 정보가 조금 이상하게 나오는 것을 볼 수 있을 것이다. C코드로 작성된 코드에 대해서는 별 문제가 없으나, 테스트 케이스 및 테스트 메인 파일, 몇몇 파일들의 커버리지 제대로 표현되지 않을 수 있다. 하지만, 단위 테스트 중인 파일(SUT)에 대한 커버리지는 정확하게 나온다.

[Unity와 CppUTest의 비교]

개인적인 성향에 따라 차이가 나겠지만, 간단하게 "Unity"와 "CppUTest"를 목(Mock)과 합쳐서 비교해 보도록 하겠다. 이것은 오로지 주관적인 결과이며 객관적인 자료는 아니다. 둘 중에 어느 것을 선택할지는 개인의 결정에 맡겨진 것이며, 단순히 설치 및 몇 가지 테스트 케이스를 작성한 결과를 토대로 했을 뿐이다. 따라서, 어떤 선입견도 없이 그냥 참고 정도의 자료로 활용하기 바란다.

1. 설치 난이도.

둘 다 쉽다. "Unity"는 다운로드 받아서 직접 프로젝트에 포함해서 사용하면 되고, "CppUTest"는 설치 스크립트를 통해서 라이브러리로 만들어서 과제에 포함시켜서 사용하면 된다.

2. 컴파일

"Unity"는 "CMock"과 사용될 경우, 다운로드 받은 디렉토리의 헤더 파일과 소스 파일들을 프로젝트로 가져와서 사용해야 한다. 반면에, "CppUTest"는 C++로 작성된 것으로 C언어와 함께 컴파일 할 경우에 각각의 언어 설정에 컴파일러의 옵션을 달리 설정해 주어야 한다. 따라서, "Unity"가 조금 더 C언어와 같이 사용하기에 편하다.

3. Mock의 생성

“Unity”는 목(Mock)을 자동으로 생성한 “Ruby” 스크립트를 가지고 있지만, “CppUTest”는 목(Mock)을 손으로 직접 작성해야 한다. 이때, 오류가 발생할 가능성이 많으며, C에서 사용하던 값들을 C++에서 지원하지 않을 가능성도 있다. 예를 들어, “NULL”이란 값을 (void*)로 타입 캐스팅을 해서 파라미터로 사용해야 하는 경우가 있다. C언어로 된 코드를 C++언어로 된 목(Mock)으로 대체하는 과정에서 생겨날 수 있는 문제들이 있을 것이라고 추측된다.

4. 테스트 케이스 작성

“Unity”는 일일이 “Test Runner”에 테스트 케이스를 추가하는 방식으로 테스트 케이스를 작성하기에, 실수로 만들어진 테스트 케이스가 추가되지 않을 가능성이 있다. “CppUTest”는 “Test Runner”的 작성은 필요치 않으나, 목(Mock)의 기대값을 설정하는 과정이 다소 복잡하다. C코딩에 익숙한 사람이 C++ 코드의 테스트 케이스 작성을 위해서는 간단한 C++문법 정도는 알아야 한다.

5. 메모리 관리

“Unity”는 목(Mock)의 메모리 관리를 할 필요가 없다. “CppUTest”的 경우에는 "mock().clear()"와 같은 것을 해주지 않을 경우, 메모리 누수(Leak)를 감지하는 프로그램에서 오류로 보고할 가능성이 있다. 따라서, "mock().clear()"를 적절히 호출하는 순간을 찾아서 테스트 케이스 작성에 넣어주어야 한다.

6. SUT(Software Under Test : Production Code) 작성

“Unity”는 C코드와 같이 컴파일 될 때 별도로 SUT의 C 헤더파일에 대한 수정이 필요 없지만, “CppUTest”는 C++에서 C함수를 호출해주어야 하기에, "#ifdef __cplusplus ~ #endif"와 같은 것이 C의 헤더파일에 추가되어야 한다. 따라서, 헤더파일의 템플릿(Template)을 새로 작성해서 적용해야 한다.

7. 기타 다른 코드의 여부

“CppUTest”는 파라미터로 전달 받는 값 들에 대한 형(Type)을 비교하기 위해서 “비교자(Comparator)”가 필요하지만, “Unity”에서는 필요없다. 만약, 복잡한 형을 파라미터로 사용한다면, 비교자를 작성하는 것이 약간의 부담이 될 수 있다(큰 부담은 아니지만).

8. 테스트 커버리지

“CppUTest”的 커버리지 데이터에는 테스트가 실패해야만 포함되는 “비교자의 “valueToString()” 메서드가 포함되어 있다. 따라서, 이 부분은 모든 테스트가 통과되고, 다른 코드들이 다 테스트가 되더라도 100% 커버리지가 되지 않는다. 조금 보기에 거슬릴 뿐 큰 문제는 아니다(단지 2라인 정도). 물론, 이것도 보기 싫다는 사람에게는 눈에 띄겠지만, 커버리지 데이터를 수집할 경우에는 이 부분은 배제해야 할 것이다.

위와 같은 이유로 다음과 같은 결론을 내릴 수 있다. C언어로 작성된 코드를 단위 테스트 하기 위해서는 “Unity”와 “CMock”을 사용하는 것이, C++언어로 작성된 코드를 단위 테스트 할 때는 “CppUTest”와 “CppUMock”을 이용하는 것이 편리할 것이라고 생각할 수 있다. 그렇다고 “CppUTest”를 C++에서만 사용할 수 있다는 이야기는 아니며, 목(Mock)을 사용하는데 다양한 메소드들을 제공하는 면과, C++로 작성해서 클래스의 메소드들을 IDE환경에서 마우스나 키보드를 이용해서 코드와 함께 볼 수 있다는 점은 장점이다. 선택은 개인의 몫이며, 어떻게 사용할 것 인가는 직접 경험해보기 바란다.

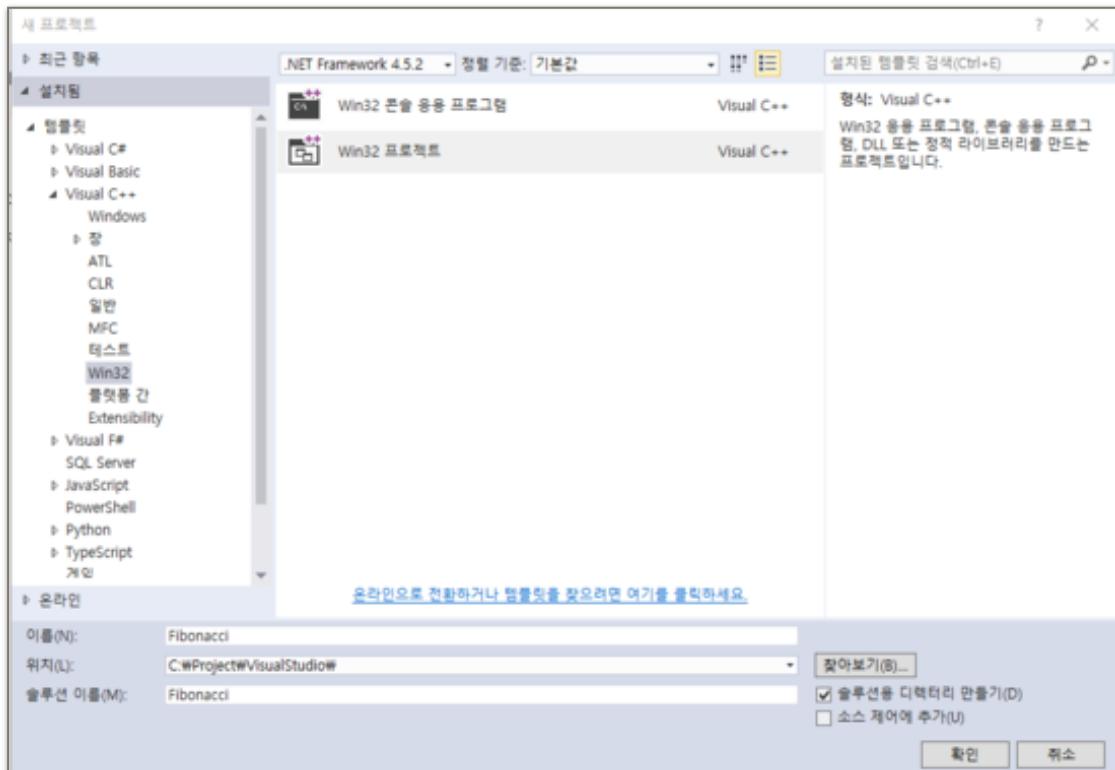
[GTest를 이용한 단위 테스트]

“GTest”는 구글에서 제공하는 C/C++ 단위 테스트 프레임워크이다. 이번에는 Eclipse가 아닌 Visual Studio를 활용한 단위 테스트를 보도록 하겠다. 여기서 사용하는 프로그램은 오픈소스로 제공되는 Visual Studio 2015 Community 버전이다. 이 프로그램은 다운로드 및 설치가 무료이기에, 윈도우와 같은 환경에서 자유롭게 사용할 수 있다. 다운로드는 아래와 같다.

<https://www.visualstudio.com/ko-kr/products/visual-studio-community-vs.aspx>

설치 시 필요한 환경이 다운로드가 되지 않았다면, Visual Studio의 "Tool(도구)"메뉴에서 "Expansion & Updates(확장 및 업데이트)" 메뉴에서 검색해서 설치할 수 있다. 나중에 커버리지의 확인을 위해서 "Open Cpp Coverage"도 찾아서 다운로드 받도록 한다.

먼저 단위 테스트 하고자 하는 프로그램의 프로젝트를 생성한다. "main()"함수가 정의되어 상관없기에 "Win32" 프로젝트로 만들도록 한다. 나중에 단위 테스트에서 정적 라이브러리로 사용 되기에 "정적 라이브러리" 옵션이 선택되어야 한다.



만약, 정적라이브러리로 설정되지 않으면, 나중에 단위 테스트 시에 단위 테스트 대상을 찾지 못하는 오류가 발생할 수 있다. 실행될 파일이 아니라도 "main()"함수를 가질 수 있기에, 여기서는 단순히 다른 프로젝트에서 참조되는 형식으로만 한정하기로 한다. 나중에 "main()"함수를 호출해서 프로그램의 시작이 될 수 있도록 프로젝트를 설정할 수 있다.

현재 단위 테스트를 위한 간단한 대상 파일을 만들기 위해서 "Project Properties(프로젝트 속성)->Add(추가)->New Item(새로운 아이템)"에서 C++ 파일이나, 헤더 파일을 추가하도록 한다. 각각은 아래와 같은 내용을 가진다.

```
/* fibonacci.h */
#pragma once
#ifndef __cplusplus
extern "C" {
#endif

unsigned long fibonacci( unsigned int index );

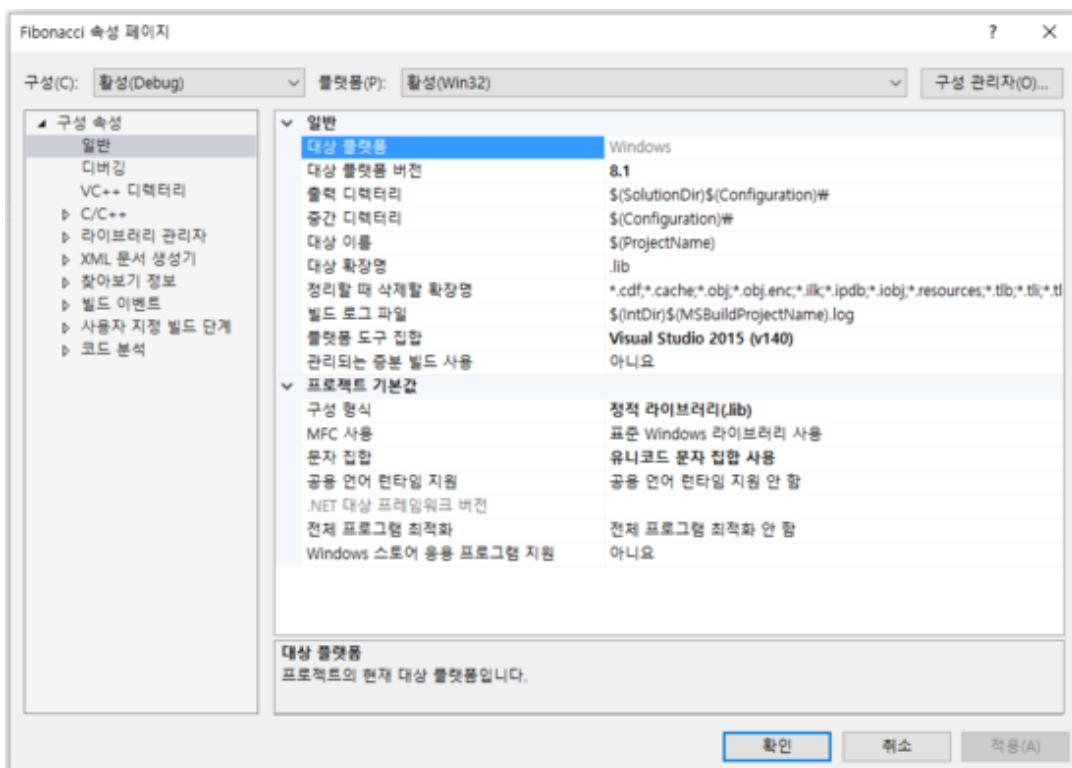
#ifndef __cplusplus
}
```

```
#endif
```

```
/* fibonacci.c */
#include "fibonacci.h"

unsigned long fibonacci( unsigned int index ) {
    if(( index == 0 ) || ( index == 1 ))
    {
        return 1;
    }
    return fibonacci( index - 2 ) + fibonacci( index - 1 );
}
```

테스트 하는 프로그램은 단순히 피보나치 수열(Fibonacci sequence)을 구하는 것이다. 재귀적인 호출을 사용해서 숫자를 구하고 있다. 첫 번째와 두 번째 원소는 지정된 값("1")을 사용한다.

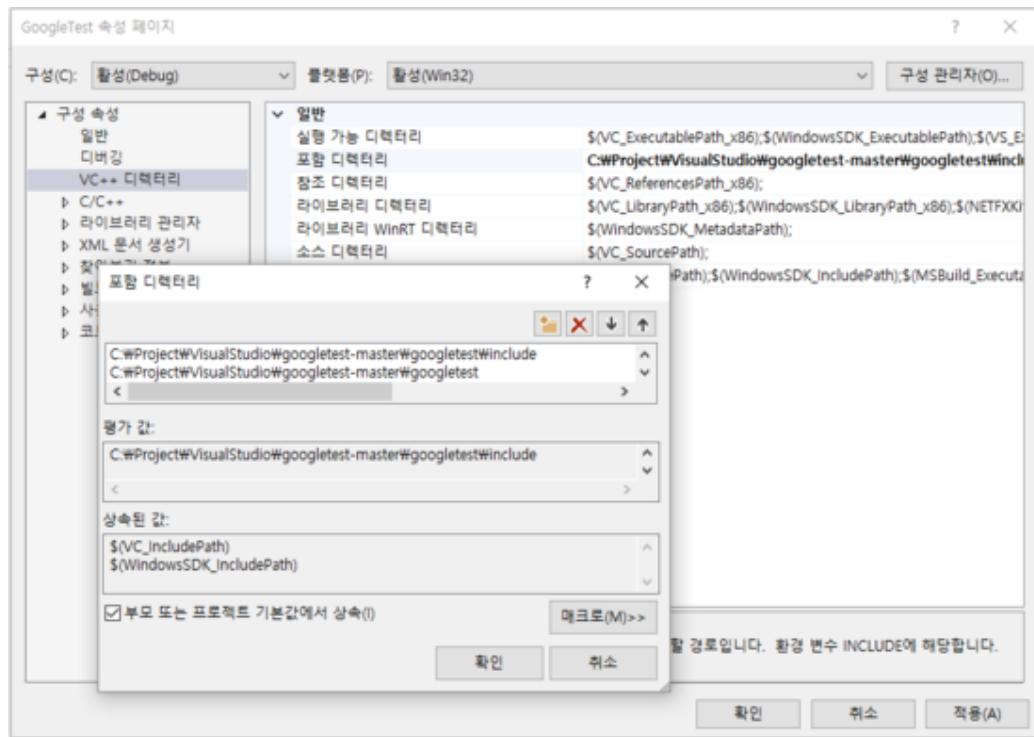


프로젝트의 속성(Properties)은 위의 그림에서 보듯이 "구성 형식"에 "정적 라이브러리(Static Library)" 주어 졌음을 확인할 수 있을 것이다. 추가적으로 프로젝트를 컴파일하기 위해서는 "Fibonacci.h"와 같은 파일이 선언된 위치를 "VC++ 디렉토리"에서 "포함할 디렉토리"에 추가해야 한다. 즉, 단위 테스트 대상이 되는 파일이 필요로 하는 "include" 디렉토리를 설정하는 것이다.

이제는 "GTest"를 위한 프로젝트를 만드는 방법을 설명하도록 하겠다. 앞에서와 마찬가지로 정적 라이브러리로 만들도록 한다. 직접적으로 실행될 이미지를 만들지 않으며, 나중에 단위 테스트를 위한 정적 라이브러리로 사용된다. "미리 컴파일된 헤더(Pre-compiled Header)"는 제외하도록 한다.

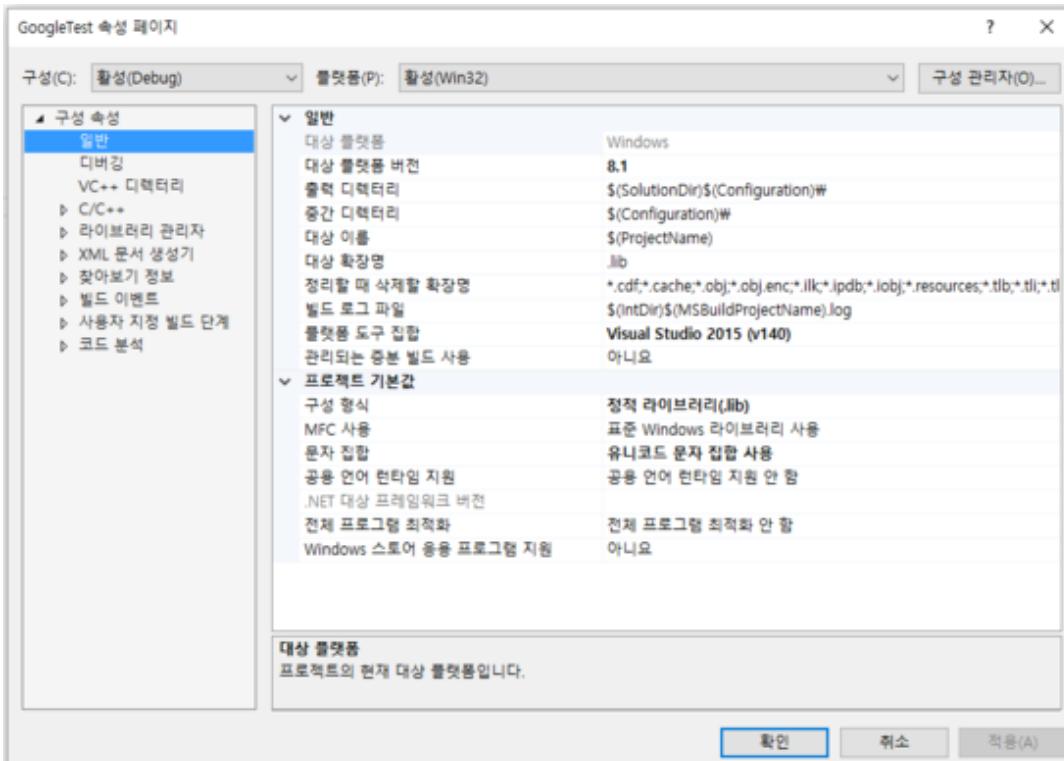


필요한 파일들은 “GTest”의 압축을 해제한 “src” 디렉토리 이하의 파일들을 전부 포함하면 된다.
"Project Properties->Add->Existing Item"을 통해서 필요한 파일들을 전부 프로젝트로 불러들인다.

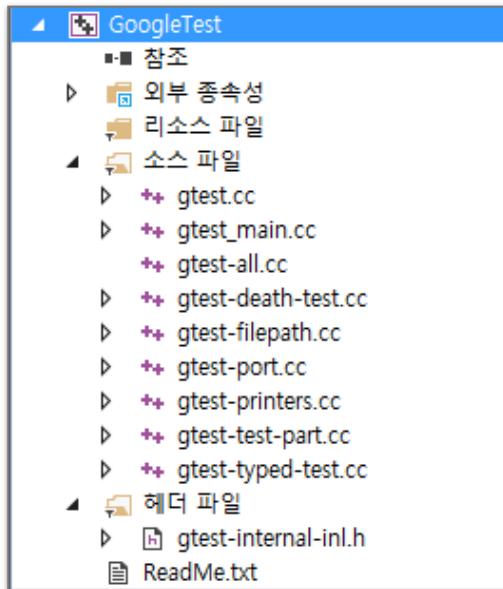


"GTest"를 빌드하기 위해서는 "Include" 디렉토리를 설정해야 한다. "VC++ 디렉토리"에서 "Include Directory"에 다음과 같은 두 개의 디렉토리를 추가한다. "GTest"의 압축을 해제한 디렉토리와 "GTest"의 "Include" 디렉토리를 찾아서 포함하도록 한다.

"GTest"도 "Static Library"로 만들어질 것 이기에, GTest 프로젝트의 속성에는 아래와 같이 "정적 라이브러리(Static Library)"로 설정되어 있어야 한다.



프로젝트에 추가된 파일들과 헤더 파일들은 아래의 그림과 같이 볼 수 있을 것이다. 지금까지 생성된 프로젝트는 단위 테스트 대상이 되는 프로젝트와 단위 테스트를 위한 프레임워크인 “GTest”이다. “GTest”的 파일들 중에서 "gtest_main.cc"는 “GTest”를 실행하기 위한 "main()" 함수를 가지지만, “GTest” 자체가 응용프로그램으로 실행될 것이 아니기에, 현재는 정적 라이브러리로 만들어 둔 것이다.



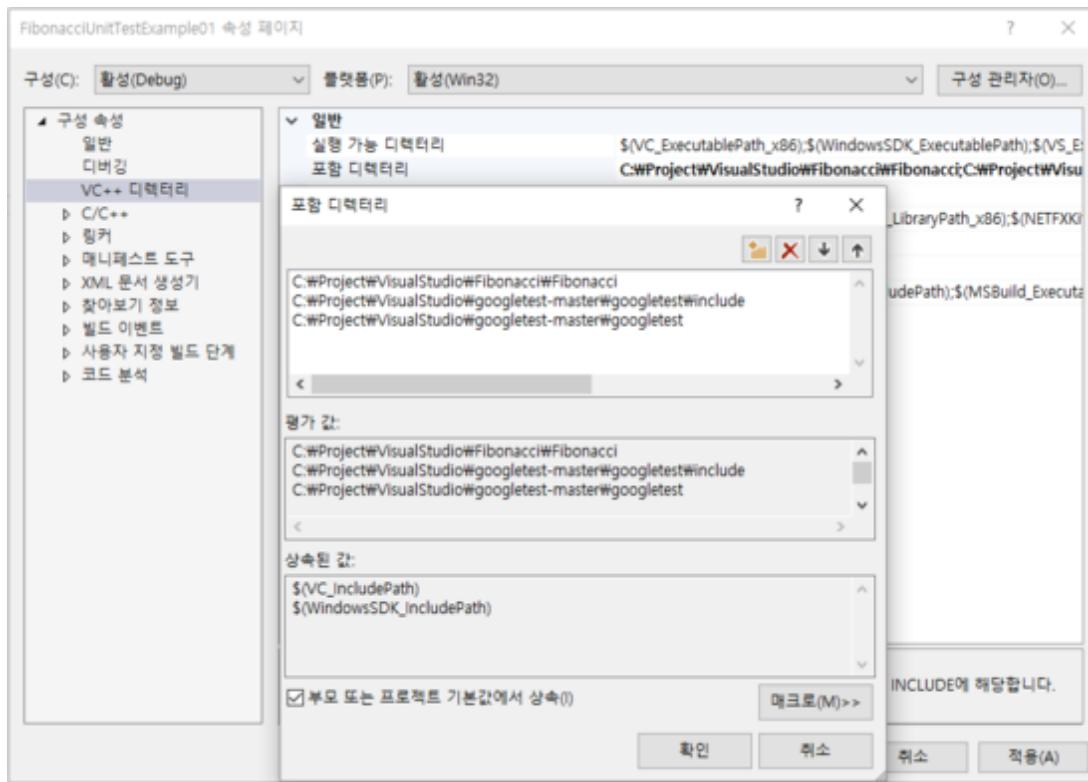
이제는 단위 테스트를 위한 테스트 케이스의 프로젝트를 만들 때다. 이번에는 실제로 실행될 프로젝트이기에 "Win32 Console Application"으로 만들어야 한다. 추가될 파일의 내용은 아래와 같다.

```
/* 일부 헤더파일의 include가 제외됨 */
TEST( Fibonacci, tenthFibonacci ) {
    EXPECT_EQ( 89, fibonacci( 10 ) );
}
```

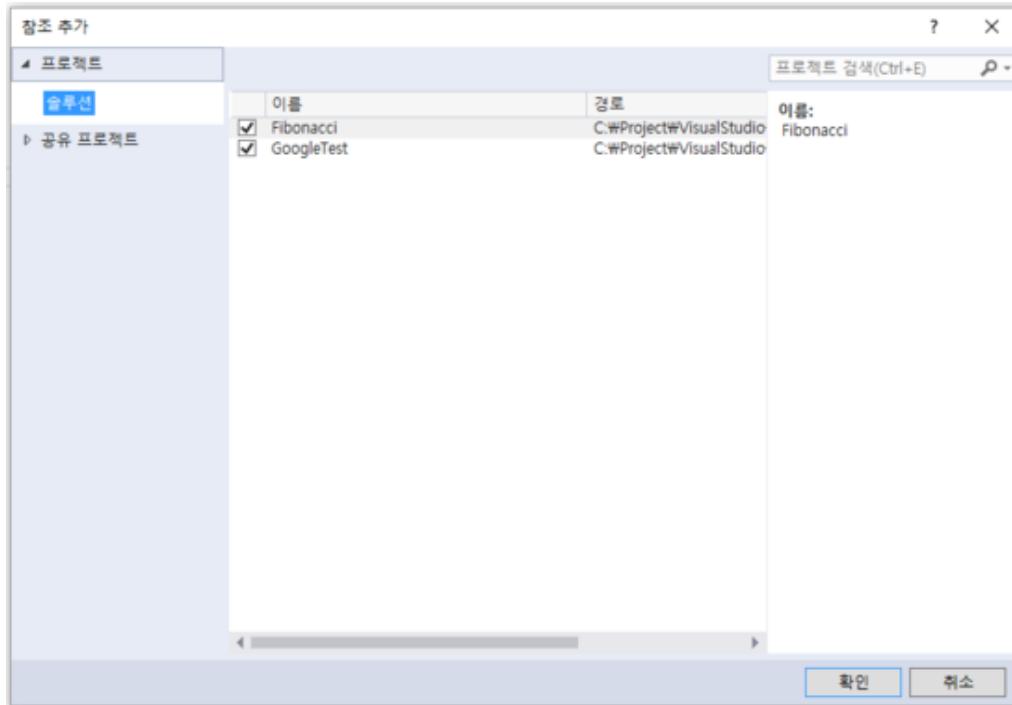
위의 테스트 케이스는 "Fibonacci"라는 테스트 그룹을 만들어서, "tenthFibonacci"라는 테스트 케이스를 추가한다는 의미이며, 테스트 내용은 10번째 피보나치 수열의 값을 구해서 비교하는 ("EXPECT_EQ()") 것이다.

만약, 테스트 대상이 되는 파일에 "main()"과 같은 것이 정의된 경우에는 단위 테스트가 실행되지 않을 수 있으니, 아래와 같은 것을 추가하도록 해야한다.

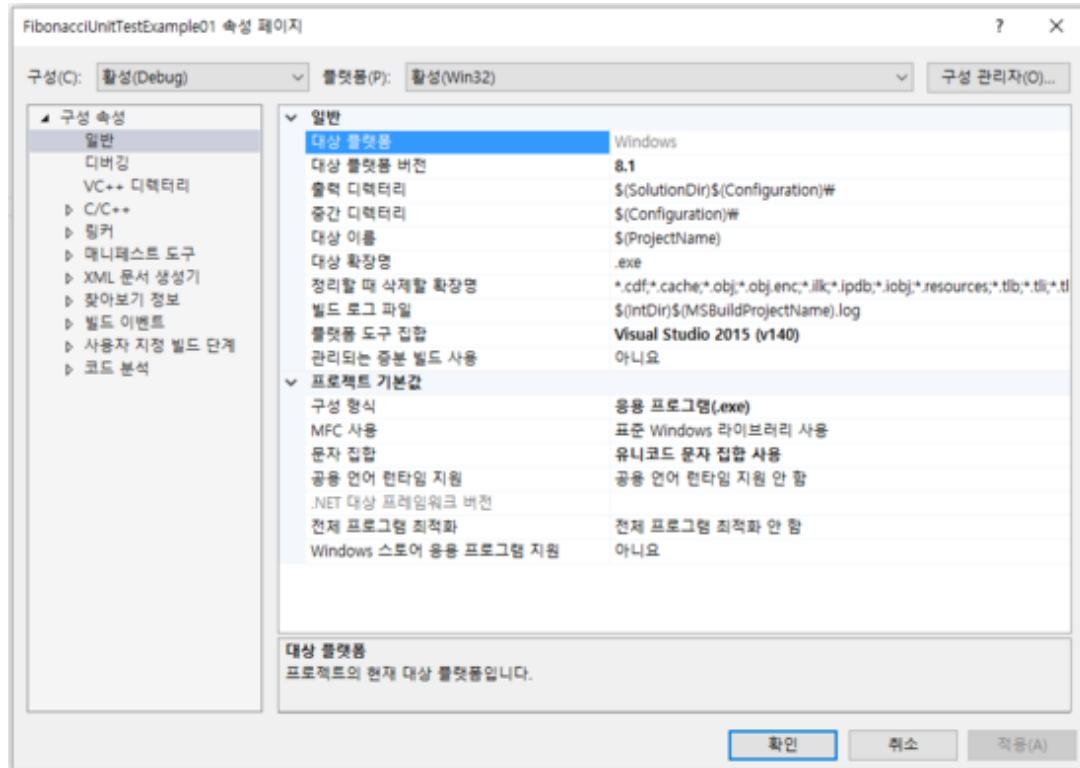
```
int main( int argc, char* argv[] ) {
    testing::InitGoogleTest( &argc, argv ); /* 구글 테스트 프레임워크의 초기화 */
    return RUN_ALL_TESTS(); /* 모든 테스트의 실행 */
}
```



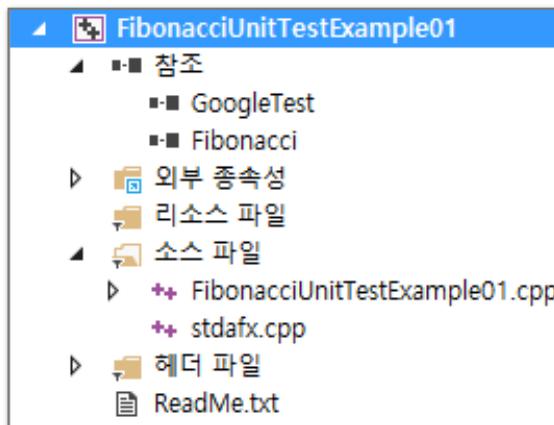
프로젝트에서 필요한 디렉토리는 앞에서 만들었던 두개의 프로젝트의 "Include" 파일들을 포함하는 것이다. 따라서, "Project->Properties"에서 "VC++ 디렉토리"에 포함될 디렉토리를 앞에서와 같은 방법으로 추가한다. 단위 테스트 대상이되는 프로젝트의 "Include"와 "GTest"의 "Include"를 추가하면 된다.



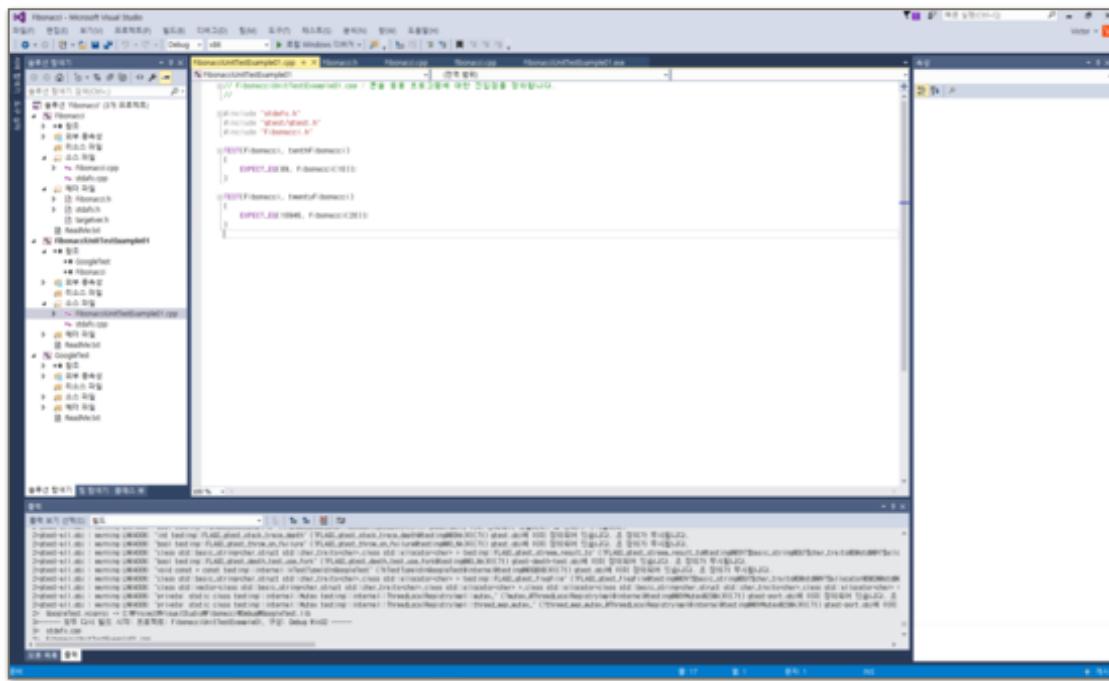
이전의 2개의 프로젝트와는 달리, 이 단위 테스트 프로젝트는 두 개의 프로젝트에 대한 참조(Reference)를 더 추가해 주어야 한다. "Project->Add->Reference.."을 선택해서 위의 그림과 같이 2개를 체크하면 된다.



이렇게 만들어진 단위 테스트 프로젝트의 설정은 위의 그림과 같이 되어 있을 것이다. 응용프로그램으로 실행될 것 이기에, 이전의 두 프로젝트와는 달리 "구성 형식(Configuration Type)"에 "응용프로그램(Application (.exe))"와 같이 되어 있어야 한다.



단위 테스트 프로젝트의 파일 및 참조 등은 위의 그림과 같이 나타날 것이다. 즉, "Google Test", "Fibonacci" 두 개의 프로젝트에 대한 참조가 추가 되었음을 알 수 있을 것이다.

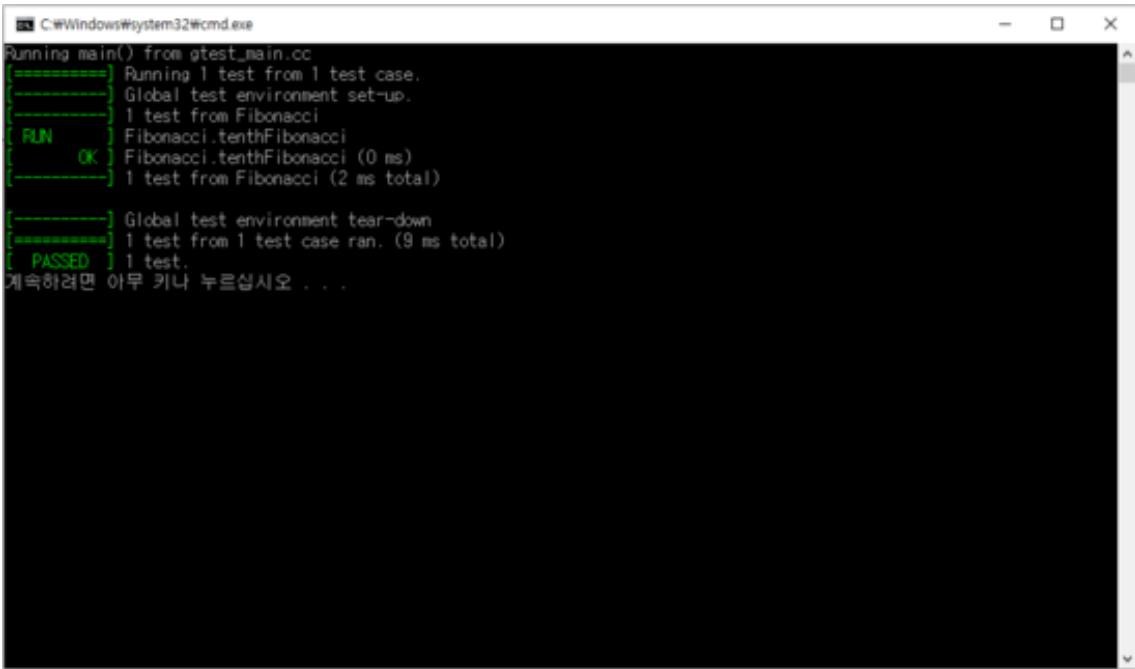


이제는 이 프로젝트를 실행해 볼 차례이다. "Ctrl-F5"를 눌러서 실행하면 된다. 실행하면 단위 테스트를 실행한 결과를 확인할 수 있을 것이다.

만약, 여기서 우리가 단위 테스트 하려고 하는 "fibonacci()" 함수와 같은 것이 "Unresolved Symbol"과 같은 오류가 발생한다면, C++ 단위 테스트 프레임 워크에서 C파일에 정의된 함수를 찾지 못하는 경우와 헤더 파일을 "Include"하지 않은 경우, 단위 테스트 대상이 되는 프로그램이 응용프로그램으로 설정되어 컴파일된 경우 세 가지를 의심해 보아야 한다. 각각에 대해서는 이미 설명한 방법대로 다시 설정을 변경하면 될 것이다.



실행한 결과는 아래와 같다. 각각의 단위 테스트 케이스의 실행한 결과(여기서는 하나만 있음, 추가해서 더 넣어도 상관없다.) 및 전체 단위 테스트 케이스의 실행한 요약 정보를 볼 수 있을 것이다. 성공했다면 녹색으로 보일 것이고, 실패한 경우에는 블은 색으로 표시될 것이다.

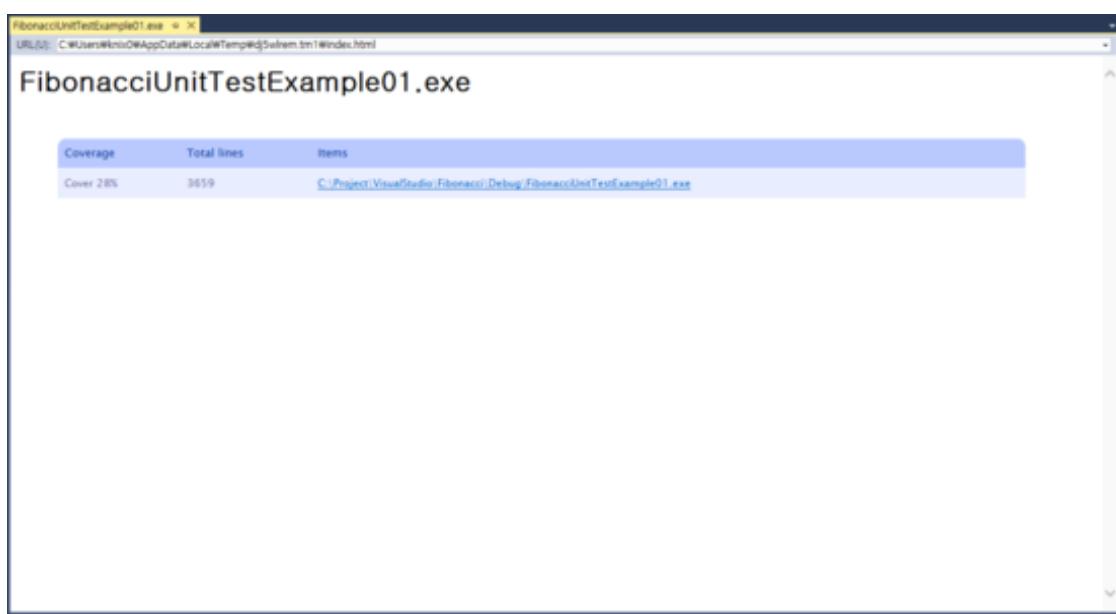


```
C:\Windows\system32\cmd.exe
Running main() from gtest_main.cc
[==========] Running 1 test from 1 test case.
[==========] Global test environment set-up.
[=====] 1 test from Fibonacci
[RUN]   Fibonacci.tenthFibonacci
[OK]    Fibonacci.tenthFibonacci (0 ms)
[=====] 1 test from Fibonacci (2 ms total)

[=====] Global test environment tear-down
[=====] 1 test from 1 test case ran. (9 ms total)
[PASSED] 1 test.

계속하려면 아무 키나 누르십시오 . . .
```

위의 실행한 결과는 녹색으로 보이기에 단위 테스트가 제대로 실행 되었다는 것을 알 수 있다. 즉, 원하는 예상 결과(Expected Value)를 단위 테스트 대상이 되는 함수가 복귀값으로 돌려 주었다는 뜻이다.



마지막으로 한가지 더 짚어봐야 하는 것은 단위 테스트 커버리지(Coverage)에 대한 것이다. 이것을 측정하기 위해서는 Visual Studio의 "Open Cpp Coverage"라는 "Plugin"이 필요하다. 설치는 "Tools(도구)->Extensions and Updates(확장 및 업데이트)"를 실행해서 찾으면 된다.

실행하기 위해서는 "Tools->Run OpenCppCoverage"를 선택하면 된다. 결과는 먼저 전체 단위 테스트에 대한 커버리지(Coverage)를 보여줄 것이다. "Items"의 링크를 열어서 각각의 파일들에 대한 커버리지를 확인할 수 있다.



각각의 파일들 중에서 우리가 관심이 있는 부분은 단위 테스트 대상이되는 함수다. 제일 아래 쪽을 보면 우리가 관심을 가지고 있는 파일들에 대한 커버리지 데이터를 볼 수 있을 것이다. 이번 예전의 경우에는 단순 하기에 100% 커버리지가 만족 됨을 볼 수 있을 것이다.

```

1. #include "stdafx.h"
2. #include "Fibonacci.h"
3.
4. unsigned long Fibonacci(unsigned int index)
5. {
6.     if ((index == 0) || (index == 1))
7.     {
8.         return 1;
9.     }
10.    return Fibonacci(index - 2) + Fibonacci(index - 1);
11. }

```

실제 실행된 프로그램의 라인을 보기 위해서는 각각의 파일을 클릭해서 들어가야 한다. "OpenCppCoverage"는 단순히 라인 커버리지(Line Coverage 혹은 Statement Coverage)만 보여주기에 적절한 커버리지가 아닐 수 있다. 즉, 분기 커버리지(Branch Coverage)나 함수 커버리지(Function Coverage)등은 찾을 수 없다. 더 나은 커버리지를 측정하기 위해서는 상업적인 툴의 사용을 검토해 볼 수 있다.

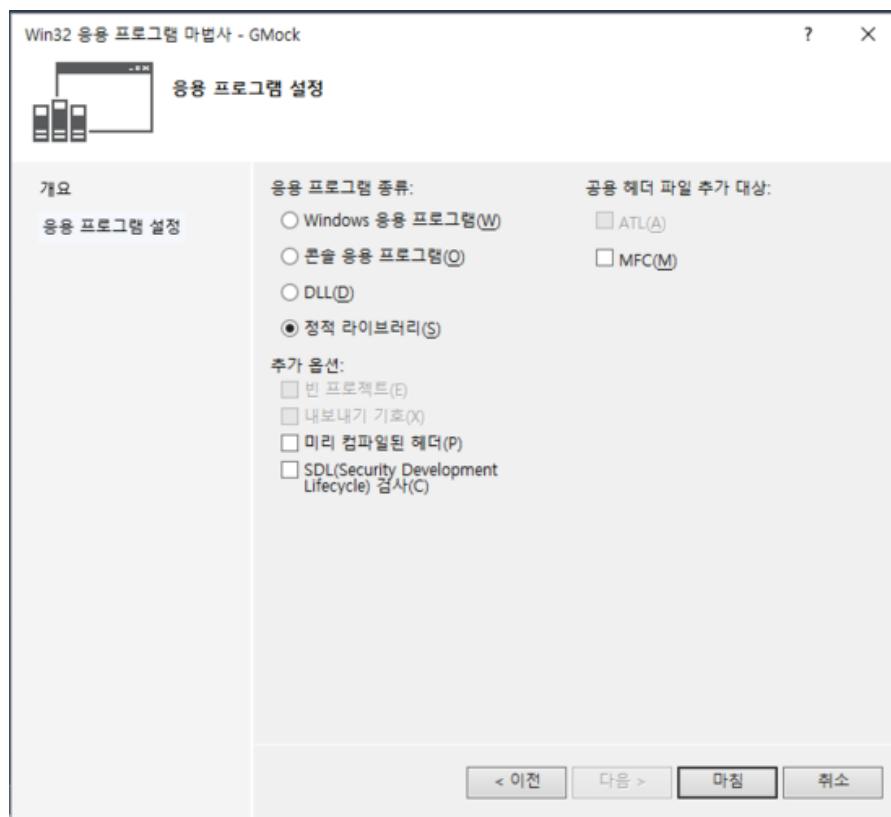
커버리지를 확인하는 것은 테스트 케이스의 보완이 필요한지 판단하기 위해서다. 즉, 테스트 되지 않은 코드를 테스트 하기 위해서는 테스트 케이스를 더 추가할 수 있다. 사실상 100% 라인 커버리지를 달성하기는 어렵기에, 대부분의 경우 95%정도의 라인 커버리지를 만족할 수 있는 수준에서 테스트 케이스를 보완하는 경우가 많다.

이상에서 “GTest”를 이용한 단위 테스트를 어떻게 하는지 살펴보았다. 여기서 더 추가할 부분은 “GTest”에서 제공하는 목(Mock)을 사용하는 방법일 것이다.

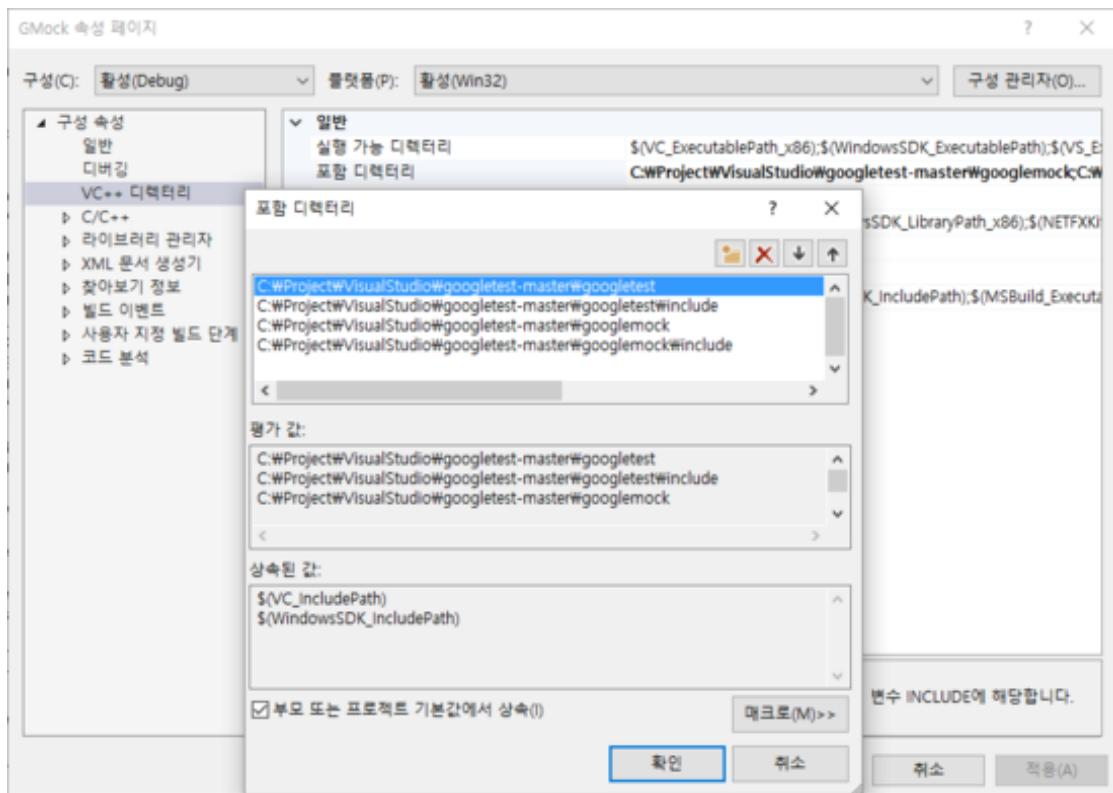
[GMock의 활용]

“GMock”은 “GTest”와 더불어 같이 배포되는 목(Mock)이다. 따라서, 기본적으로 “GTest”를 사용한다면 같이 사용해야 할 경우가 많을 것이다. “GTest” 자체가 C++로 만들어져 있어서, “GMock”도 C++로 되어있다. C언어에서 이를 사용하기 위해서는 C 단위 테스트에서 필요한 함수들을 위한 목(Mock) 역할을 하는 객체를 생성해서, 메소드를 사용해서 구현하는 방식으로 이용하게 된다. 인터넷에서는 C언어를 위한 “GMock”的 활용에 대해서는 많은 자료를 찾아보기 힘들기에, 일단 여기에서 제공하는 방법을 익히고, 더 좋은 방법을 찾아보면 될 것이다.

“GMock”도 라이브러리 형식으로 프로젝트를 생성해야 한다. 생성하는 방법은 앞에서 “GTest” 프로젝트를 만드는 것과 동일하다. 정적 라이브러리 형식으로 생성하기에, “File->New->Project”에서 “GMock” 프로젝트를 생성한다.



실행되는 프로그램이 아니기에, "Win32 프로그램"으로 프로젝트의 속성을 설정하고, 정적 라이브러리(Static Library)로 체크해 주어야 한다. "미리 컴파일된 헤더(Pre-compiled Header)"는 설정하지 않아도 된다.



"GMock" 프로젝트의 파일들은 "Google Test"의 압축을 해제한 디렉토리에 있는 "Google Mock"에서 "src" 디렉토리를 그대로 가져오면 된다. "Project->Add->New Item->File System"과 같이 선택해서, 찾으면 될 것이다. 디렉토리를 분리하고자 한다면, 새로운 디렉토리를 프로젝트에 생성하고 그곳으로 옮겨 넣어주면 된다.

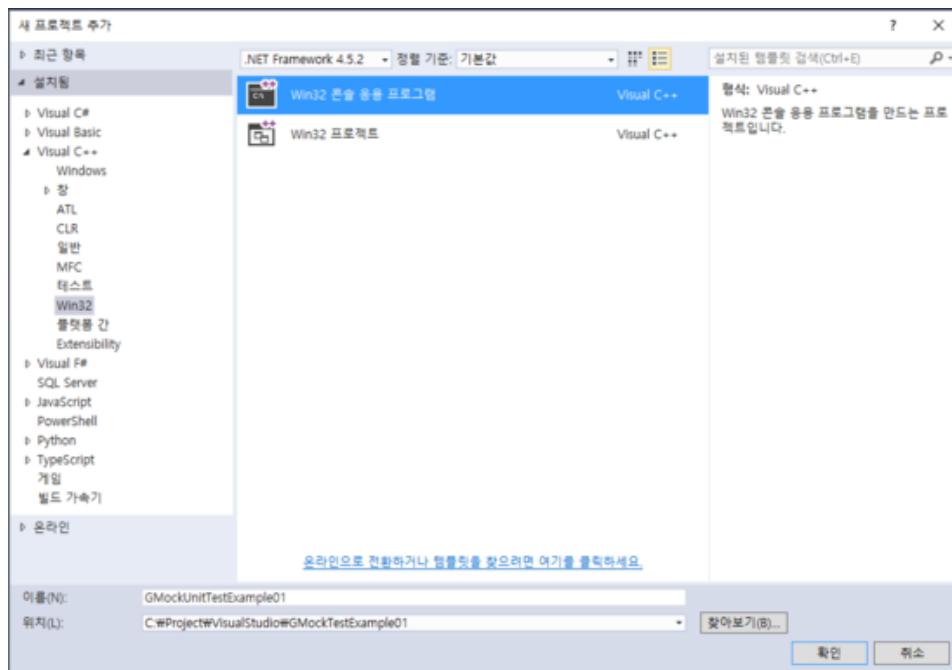
컴파일을 하기 위해서는 "Include" 디렉토리를 찾아야 하기에, "Project->Properties->Configuration Properties->VC++ Directory"에서 "GMock"을 위해서 아래와 같은 4개의 디렉토리를 추가하도록 한다.

1. Google Test의 디렉토리
2. Google Test의 "Include" 디렉토리
3. Google Mock의 디렉토리
4. Google Mock의 "Include" 디렉토리

압축을 해제한 디렉토리가 필요한 이유는 소스 코드 내에서 "#include xxx.cc"와 같이 소스 코드를 포함하는 경우가 있기 때문이다.

이상과 같이 했다면, 컴파일을 통해서 제대로 오류가 없이 빌드(Build)가 되는지를 확인해야 할 것이다. 빌드가 되지 않는다면, 오류 메시지에 따라 프로젝트의 일부 설정을 변경해야 할 수도 있다.

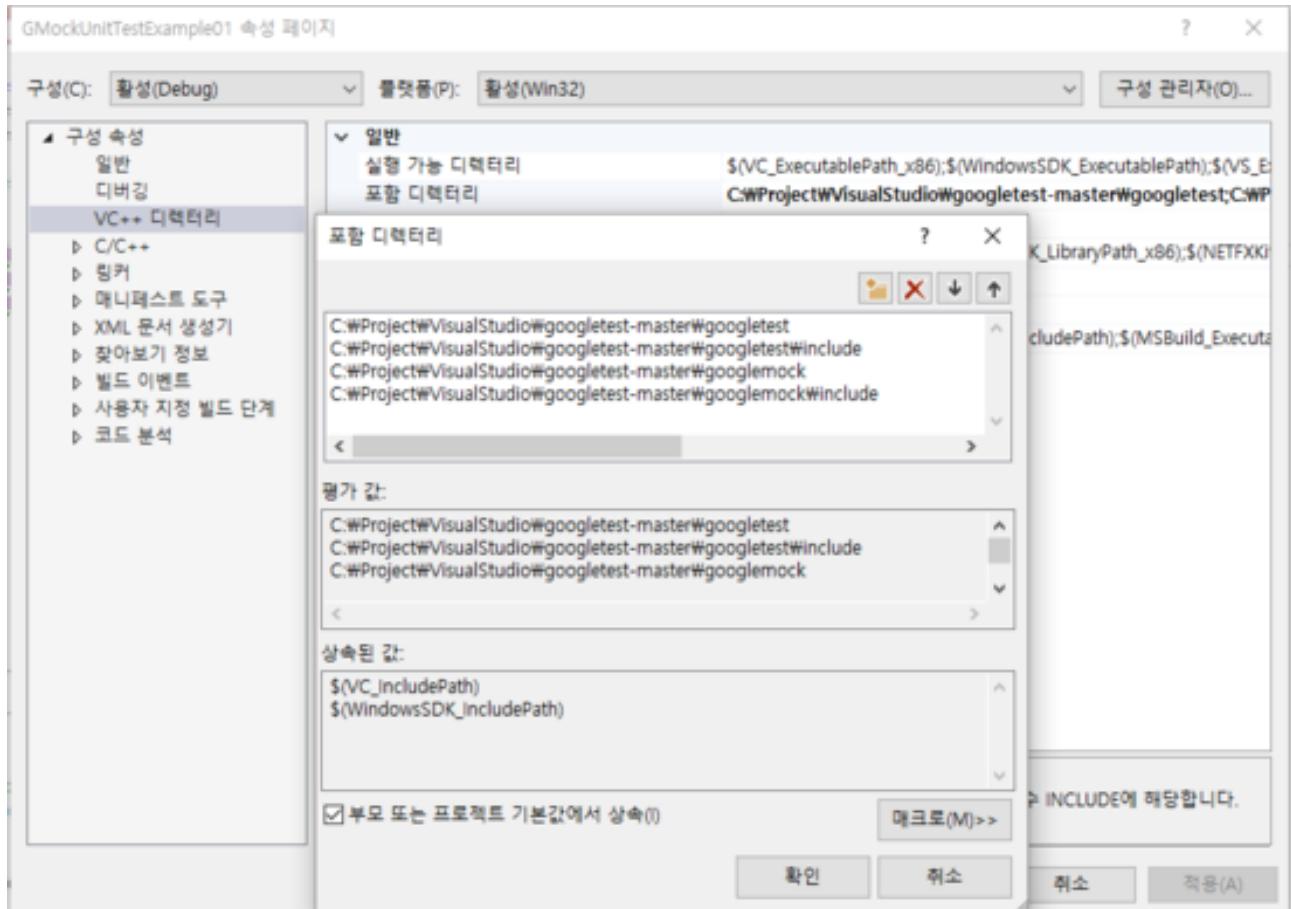
"GTest"에 대한 프로젝트를 만들고 빌드하는 과정은 앞에서 이미 이야기했기 때문에 생략한다. 다음은 실제 단위 테스트 작성할 차례이다. 이를 위해서는 당연히 "Win32 Console Application"으로 프로젝트를 생성해야 할 것이다. 목(Mock)을 사용하는 테스트를 할 것 이기에, 일부 코드들을 미리 작성해서 보여줄 것이다.



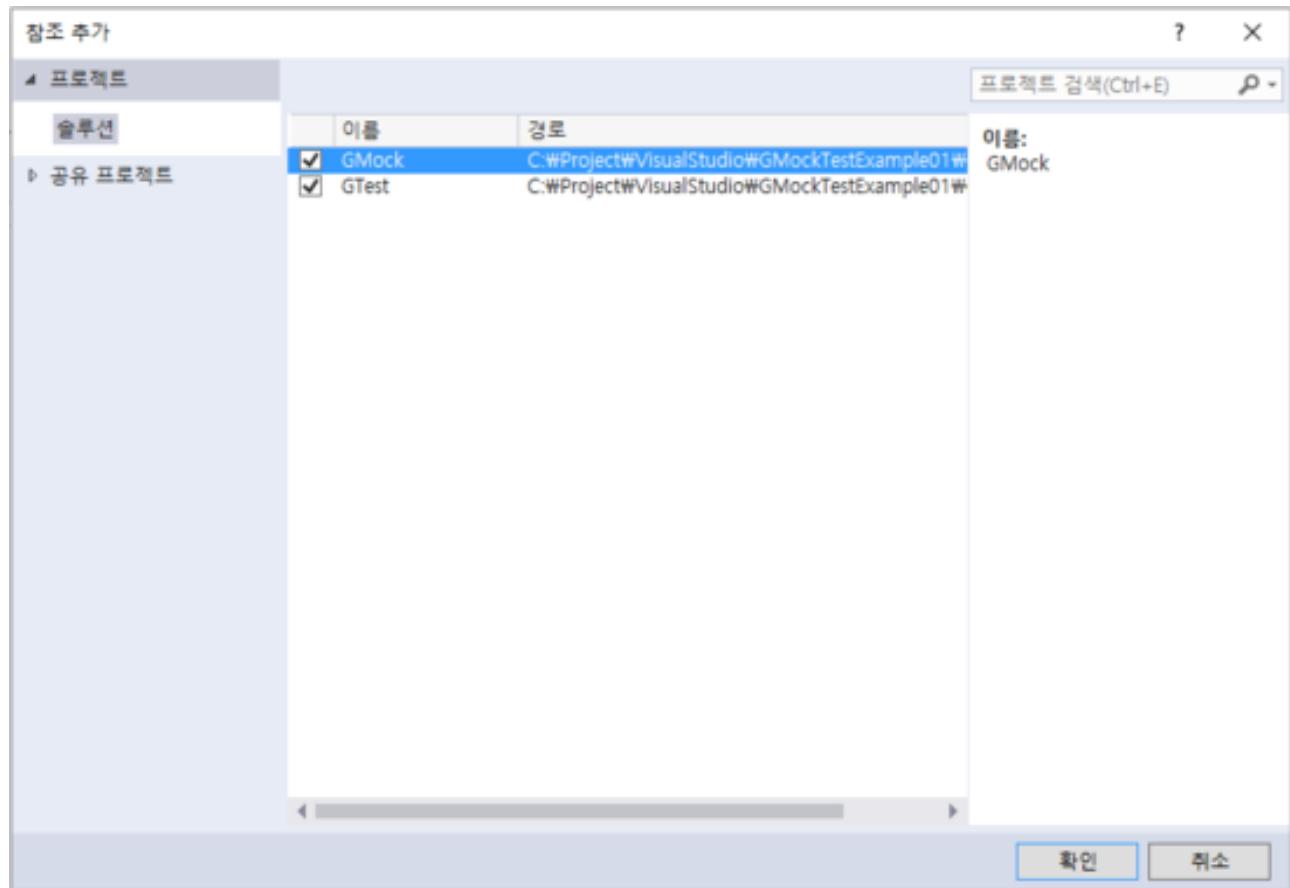
C언어로 작성된 코드들은 C++ 컴파일러를 이용해서도 빌드가 가능하다. 하지만, “GTest”와 “GMock”에서 C 함수들을 접근하는 것과, C로 작성된 코드에서 C++로 작성된 “GTest”와 “GMock”을 사용하는 것은 어렵다. 따라서, 일단은 C로 작성된 함수들을 마치 C++로 작성된 함수로 취급하도록 한다.



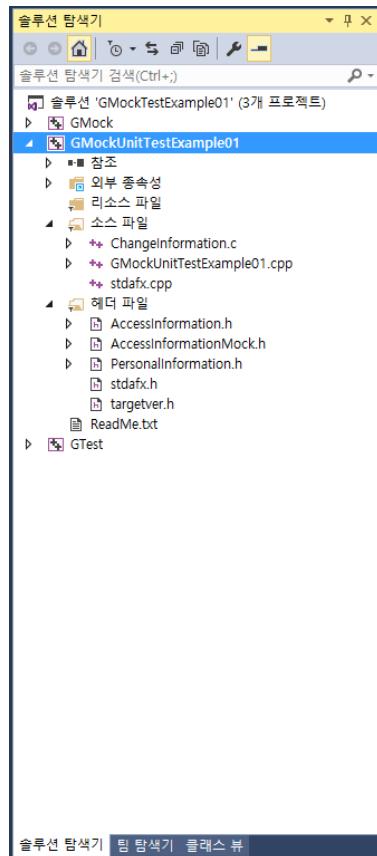
단위 테스트를 위한 프로젝트 설정에도 "미리 컴파일된 헤더(Pre-Compiled Header)"옵션을 사용하지 않도록 한다. 만약, 관련된 오류 메시지가 프로젝트 빌드에서 발생한다면, 프로젝트 설정을 다시 수정할 수 있기에, 여기서는 선택하지 않는다.



단위 테스트에서도 필요한 "Include" 파일들이 있기에, "GMock"과 같이 "GMock"의 "Include" 디렉토리들과 "GTest"의 "Include" 디렉토리들을 전부 추가하도록 한다. 아직 까지 코드를 작성하는 것은 아니고, 단순히 프로젝트를 위한 환경만 구축한 상태다.



이제는 앞서 만들었던, 프로젝트들을 참고할 차례다. 즉, “GTest”와 “GMock” 두 개의 프로젝트를 “Project->Add->Reference”에서 체크해 주면 될 것이다. 위의 그림과 같이 나타나지 않은 경우는 현재 만들고 있는 솔루션(Solution)에 해당 프로젝트가 없기 때문이다. 필요하다면 이미 만들어진 “GTest”와 “GMock” 프로젝트를 솔루션에 추가해 주어야 할 것이다.



최종적으로 만들어진 프로젝트들은 "GMock", "GTest", "GMock"과 "GTest"를 이용한 단위 테스트 프로젝트 3개가 될 것이다. 위의 그림에서 보듯이 현재는 단위 테스트와 단위 테스트의 대상이 되는 코드를 분리하지 않았지만, 각각을 분리해서 관리하는 것이 나중을 위해서 편할 것이다. 여기서는 편의상 같은 디렉토리에 둘 다 두었다. 테스트 케이스의 수가 많아지면, 코드의 물리적인 구성과 관리에 신경 써야 한다.

```
/* PersonalInformation.h */
#pragma once
#include "AccessInformation.h"

#ifndef __cplusplus
extern "C" {
#endif

bool change_information(INFO *info);

#ifndef __cplusplus
}
#endif

#endif
```

"#pragma once"는 표준은 아니지만, 다양한 컴파일러에서 헤더 파일이 중복으로 정의되는 것을 막아 준다. 현재 단위 테스트의 대상이 되는 함수는 "change_information()"이라는 함수이다. 이 파일은 단위 테스트의 대상이 되는 함수만 따로 분리한 것으로 가정해서 만들었다.

```
/* ChangeInformation.c */
#include <string.h>
#include "AccessInformation.h"
```

```

bool change_information(INFO *info) {
    INFO *oldinfo;

    /* Check parameter */
    if (info == NULL) {
        return false;
    }

    /* Print old information */
    if ((oldinfo = get_information()) == NULL) {
        return false;
    }
    print_information(oldinfo);

    /* Set new information */
    if (set_information(info) == NULL) {
        return false;
    }
    return true;
}

```

앞에서 보았던 예제와 동일하게 실제로 구현된 "change_information()" 함수는 내부적으로 3개의 함수에 의존적이다. 각각은 "get_information()", "print_information()", "set_information()"이다. 이들 각각의 함수는 아직 코딩(구현)이 안된 함수 이기에, "change_information()" 함수를 테스트하기 위해서는 목(Mock)으로 대체되어야 한다.

```

/* AccessInformation.h */
#pragma once
#include <stdbool.h>

#ifndef __cplusplus
extern "C" {
#endif

typedef struct PersonalInfo {
    char *name;
    int age;
    bool alive;
}INFO;

INFO *get_information();
INFO *set_information(INFO *info);
void print_information(INFO *info);

#ifndef __cplusplus
}
#endif

```

목(Mock)으로 대체될 함수와 필요한 자료구조를 정의한 파일이다. 코드에서 보듯이 단순한 C로 작성된 헤더 파일이다. C++에서 함수의 심볼(Symbol)을 찾을 수 있도록 하기 위해서 "#ifdef __cplusplus"

extern "C" { ... } #end"를 사용했다.

```
/* AccessInformationMock.h */
#pragma once
#include "AccessInformation.h"

class AccessInformation {
public:
    virtual ~AccessInformation() {};
    virtual INFO* get_information() = 0;
    virtual INFO* set_information(INFO* info) = 0;
    virtual void print_information(INFO* info) = 0;
};
```

목(Mock)을 위한 헤더 파일에는 목(Mock)이 상속할 클래스(Class)를 정의해야 한다. 특히, 파괴자(Destructor)는 반드시 "virtual"로 선언되어 있어야 한다. 나머지 목(Mock)으로 대체할 함수들도 멤버 함수로 "virtual"로 정의한다. 나중에 여기서 정의된 함수들은 실제 목(Mock) 클래스에서 반드시 재 정의할 것이기 때문이다.

```
/* GMockUnitTestExample01.cpp */
#include "gmock/gmock.h"      /* GMock의 매크로들을 위해서 필요함 */
#include "gtest/gtest.h"       /* GTest의 매크로들을 위해서 필요함 */
#include "PersonalInformation.h"
#include "AccessInformationMock.h"

using ::testing::AtLeast;    /* Times()에서 사용하기 위해서 필요함 */
using ::testing::Return;     /* Return()에서 사용하기 위해서 필요함 */

class AccessInformationMock : public AccessInformation {
public:
    /* 상속 받은 Mock 클래스에서 재정의 하고 있다. */
    MOCK_METHOD0(get_information, INFO*());
    MOCK_METHOD1(set_information, INFO*(INFO*));
    MOCK_METHOD1(print_information, void(INFO*));
};
```

목(Mock)의 클래스 정의를 파일로 분리해도 되지만, 구현의 편의로 그냥 같이 단위 테스트 케이스에 넣어서 정의했다. 앞에서 정의한 목(Mock)의 기본 클래스를 "public"으로 상속한다. 나머지 함수들도 같이 상속되어 여기서 실제로 정의해 주게 된다. 정의는 파라미터의 수에 해당하는 "MOCK_METHODn()"을 이용한다. "n"은 필요한 파라미터의 수다. 목(Mock)으로 대체할 함수의 이름과 리턴 타입, 파라미터의 타입만 표시한다.

```
AccessInformationMock *acMock;

INFO* get_information() {
    return acMock->get_information();
}

INFO* set_information(INFO* info) {
    return acMock->set_information(info);
}
```

```
void print_information(INFO* info) {
    acMock->print_information(info);
}
```

이제는 C에서 목(Mock)을 접근하기 위한 방법을 만들 차례다. C로 작성된 코드를 고쳐서는 안되기에(완성된 프로그램에 들어가야 하기에, “Production Code”), 여기서는 일종의 래퍼(Wrapper) 역할을 하는 대체 함수를 만들었다. 내부에서는 목(Mock)의 함수들을 호출하는 형식으로 구현되었다. 목(Mock)을 포인터로 만든 이유는 각각의 테스트 케이스마다 독립적인 실행이 보장되도록 만들어주었기 때문이다.

```
// Test Fixture.
class PersonallInfoTest : public testing::Test {
public:
    void SetUp()
    {
        acMock = new AccessInformationMock();
    }

    void TearDown()
    {
        delete acMock;
    }
};
```

테스트 픽스처(Test Fixture)는 테스트 케이스를 실행하기 위한 환경을 만들어주는 역할을 한다. 테스트 픽스처 클래스의 이름과 동일한 테스트 케이스들은, 실행 시에 테스트 픽스처에서 정의한 "Setup()"과 "TearDown()"를 실행 전과 실행 후에 각각 자동으로 호출해 준다. 여기서는 목(Mock)을 생성하고, 다시 해제하는 것을 두었다. 따라서, 각각의 테스트 케이스는 전용 목(Mock)의 생성을 보장 받는다. 테스트 픽스처를 만들기 위해서는 "testing::Test"를 "public"으로 상속 받아야 한다.

```
TEST_F(PersonallInfoTest, changePersonallInformationSuccess) {
    INFO oldInfo = { "MJ Yoon", 45, true };
    INFO newInfo = { "SH Kwon", 46, true };

    EXPECT_CALL(*acMock, get_information()).Times(1).WillOnce(Return(&oldInfo));
    EXPECT_CALL(*acMock, print_information(&oldInfo)).Times(1);
    EXPECT_CALL(*acMock,
               set_information(&newInfo)).Times(1).WillOnce(Return(&oldInfo));
    EXPECT_EQ(true, change_information(&newInfo));
}
```

앞에서 사용했던 단위 테스트와는 달리, 여기서는 테스트 픽스처를 사용하기에 "TEST_F()"를 사용해서 테스트 케이스를 정의했다. 첫 번째 나오는 이름은 반드시 앞에서 정의한 테스트 픽스처의 클래스 이름과 같아야 한다. 두 번째 나오는 이름은 테스트 케이스의 이름으로 테스트 케이스에 대한 묘사적인 이름으로 정해주면 된다.

C++컴파일러에 따라, "INFO oldInfo={ "MJ Yoon" ... };"에서 경고(Warning) 메시지를 내는 경우도 있다. 이때는 문자열 앞에 명확한 타입 캐스팅을 추가하도록 하자. 여기서는 "(char*)"을 문자열 앞에 추

가하면 된다. 만약, C에서 사용하는 함수들의 심볼(Symbol)을 찾을 수 없다는 링커(Linker) 오류가 발생한다면, "#ifdef __cplusplus extern "C" { #endif #ifdef __cplusplus } #endif"를 이용해서 함수들과 자료구조 헤더 파일까지 포함시켜주도록 하면 해결 될 것이다.

먼저, 목(Mock, “acMock”)에 대한 설정을 하기 위해서 어떤 함수가 몇 번("Times()") 호출될 것인지를 설정하고, 그 때 어떤 값을 돌려줄지("WillOnce(Return())")를 미리 알려주었다("EXPECT_CALL()"). 여기서 한 가지 주의해야 할 점은 실제 함수가 호출되는 코드의 문자열(Spelling)을 그대로 가져와야 한다는 점이다. 예를 들어, "set_information(&newInfo)"와 같이 코드에서 사용되는 것을 그대로 가져왔다. 만약, 이것을 해주지 않으면, 실제로 호출은 일어나지만, 단위 테스트 결과에서는 호출되지 않았다고 표시될 것이다.

```
TEST_F(PersonalInfoTest, changePersonalInformationWithNULLParameter) {
    EXPECT_EQ(false, change_information(NULL));
}
```

이 테스트 케이스는 "change_information()" 함수가 "NULL" 파라미터를 가지고 호출된 경우를 테스트하기 위한 것이다. 그때는 "false"를 돌려주어야 하기에, "EXPECT_EQ()"를 가지고 복귀 값을 비교했다.

```
TEST_F(PersonalInfoTest, getInformationReturnNULL) {
    INFO newInfo = { "SH Kwon", 46, true };
    EXPECT_CALL(*acMock, get_information()).Times(1).WillOnce(Return(nullptr));
    EXPECT_EQ(false, change_information(&newInfo));
}
```

이 테스트 케이스는 "get_information()" 함수가 "NULL"을 돌려주는 경우를 위한 것이다. 이때도 마찬가지로 "change_information()" 함수는 "false"를 돌려주어야 한다.

```
TEST_F(PersonalInfoTest, setInformationReturnNULL) {
    INFO oldInfo = { "MJ Yoon", 45, true };
    INFO newInfo = { "SH Kwon", 46, true };

    EXPECT_CALL(*acMock, get_information()).Times(1).WillOnce(Return(&oldInfo));
    EXPECT_CALL(*acMock, print_information(&oldInfo)).Times(1);
    EXPECT_CALL(*acMock,
               set_information(&newInfo)).Times(1).WillOnce(Return(nullptr));
    EXPECT_EQ(false, change_information(&newInfo));
}
```

이 테스트 케이스는 "set_information()" 함수가 "NULL"값을 돌려줄 경우를 위한 것이다. 이때도 마찬가지로 "change_information()" 함수는 "false"를 돌려주어야 한다.

"EXPECT_CALL()"이나 "MOCK_METHODn()"과 같은 것에 대한 정보를 더 자세히 알고 싶다면, "GTest"에서 제공하는 문서나 웹 페이지를 참고하기 바란다. 여기서는 간단한 사용법 만을 확인하기 위해서 자세한 내용은 제공하지 않는다.

```
int main(int argc, char* argv[]) {
    // You don't need to initialize Google Test. GMock will do it.
    ::testing::InitGoogleMock(&argc, argv);
    return RUN_ALL_TESTS();
```

```
}
```

마지막으로, 단위 테스트를 실행하기 위해서 "main()"함수를 만들어 주었다. "GTest"와는 달리, "initGoogleMock()"을 호출하고, "RUN_ALL_TESTS()"를 사용했다. 단위 테스트를 실행("Ctrl-F5") 한 결과는 다음과 같다.

```
C:\Windows\system32\cmd.exe
[ RUN      ] PersonalInfoTest.changePersonalInformationSuccess
[ OK       ] PersonalInfoTest.changePersonalInformationSuccess (2 ms)
[ RUN      ] PersonalInfoTest.changePersonalInformationWithNULLParameter
[ OK       ] PersonalInfoTest.changePersonalInformationWithNULLParameter (0 ms)
[ RUN      ] PersonalInfoTest.getInformationReturnNULL
[ OK       ] PersonalInfoTest.getInformationReturnNULL (0 ms)
[ RUN      ] PersonalInfoTest.setInformationReturnNULL
[ OK       ] PersonalInfoTest.setInformationReturnNULL (1 ms)
[ RUN      ] PersonalInfoTest.setInformationReturnNULL (1 ms)
[ RUN      ] 4 tests from PersonalInfoTest (11 ms total)

[ Global test environment tear-down ]
[ RUN      ] 4 tests from 1 test case ran. (16 ms total)
[PASSED   ] 4 tests.

계속하려면 아무 키나 누르십시오 . . .
```

그림에서 보듯이, 4개의 테스트를 실행 했고, 전부 성공 했음을 알 수 있을 것이다. 테스트 학습처의 "Setup()"과 "TearDown()"에 "cout<<"xxx"<<endl"과 같은 것을 추가해서, 정말로 각각의 테스트 케이스마다 한번씩 실행되는지를 확인할 수 있을 것이다.

```
1. #include <iostream.h>
2. #include "AccessInformation.h"
3.
4. bool change_information(INFO *info)
5. {
6.     INFO *oldinfo;
7.
8.     // Check parameter
9.     if (info == NULL)
10.    {
11.        return false;
12.    }
13.
14.    // Print old information
15.    if ((oldinfo = get_information()) == NULL)
16.    {
17.        return false;
18.    }
19.    print_information(oldinfo);
20.
21.    // Set new information
22.    if (set_information(info) == NULL)
23.    {
24.        return false;
25.    }
26.    return true;
27. }
```

테스트 커버리지는 테스트 케이스 각각을 전부 수행한 결과를 이용해서 어떤 코드들이 실행 되었는지 "녹색"으로 표시해 줄 것이다. 이를 위해서는 "Tools->Run OpenCppCoverage"를 실행해야 한다. 물론, 설치되지 않았다면, "Plugin"을 찾아서 설치한 후에 실행해보도록 하자.

이상에서 간단히 "GTest"와 "GMock"을 이용한 C언어 단위 테스트를 어떻게 하는지 살펴보았다. 단위 테스트는 코드 리뷰와 더불어 가장 많이 버그를 사전에 걸러 낼 수 있는 방법이기에, 반드시 개발자가 알고 있어야 할 기술이다. 또한, 테스트 자동화와 연계해서 안정적인 코드를 항상 유지할 수 있는 방법이기에, 전체 프로젝트를 성공으로 이끄는 도구를 제공해 줄 것이다.

단위 테스트는 TDD(Test Driven Development)와 같은 같은 곳에서도 사용되며, Agile 방법론을 실행하기 위해서도 필요하다. 단위 테스트를 만들기 위해서는 설계가 선행되어야 하며, 설계에서 필요한 부분은 의존하고 있는 코드 들에 대한 인터페이스 정의다. 즉, 인터페이스 정의가 있다면 목(Mock)을 활용해서 구현되지 않은 코드에 의존하는 코드도 단위 테스트를 할 수 있다. 물론, 인터페이스가 없다면 미리 정의하고 나중에 그것을 의존하고 있는 해당 인터페이스에 포팅하는 것도 생각해 볼 수 있다.

[참고] 현재는 MinGW를 이용해서 Windows에서 Eclipse를 이용한 Google Test 단위 테스트 환경을 구축하는 것은 문제가 있는 것으로 생각된다. 특히, 쓰레드(Thread)와 관련된 부분에서 문제가 발생하는데, 아직 수정되지 않은 것으로 보인다(2016.06.09 현재).

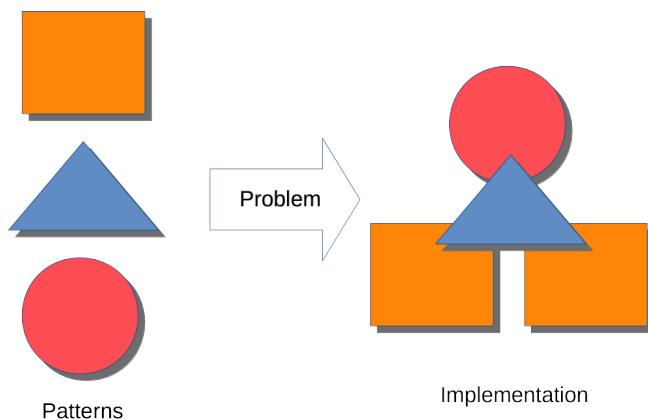
[참고] Linux Ubuntu 16.04에서는 Eclipse Mars버전이 GTK 문제로 조금 고생은 했지만, 정상적으로 동작하는 것을 확인했다. C코드를 g++컴파일이 안되고, gcc를 기본으로 설정해서 컴파일 되면서 몇 가지 경고 메시지는 나온다. Mac OS X El capitan버전의 Eclipse에서는 아무 문제 없이 실행된다.

6. 디자인 패턴

C언어는 객체지향 언어가 아니다. 객체지향 언어가 가지는 상속이나 다형성(Polymorphism)과 같은 기능을 제공하지 않으며, 서브 클래싱(Sub-classing)을 이용한 확장도 당연히 안된다. 하지만, 그렇다고 디자인 패턴에서 제시하는 아이디어들을 사용하지 못할 이유가 없다. 좋은 아이디어는 언어와 상관없이 차용 되기에 충분하다. 제약은 있지만 그렇다고 아이디어가 완전히 무관 하지는 않다. 따라서, C언어라고 해서 구현과 인터페이스를 분리해서 상세한 구현을 숨기는 것과 같은 것을 안할 이유는 없다.

C언어는 배우기 쉽고 사용하기 쉽다. 물론, 포인터를 사용하는 초보자에게 어려운 일이지만, 그나마도 객체지향 언어보다는 코딩하는데 까지 걸리는 시간이 짧다. 그러다보니 소프트웨어를 전문적으로 배운 경험이 없는 사람들도 코딩을 할 수 있고, 하드웨어를 제어하는데는 아직까지 C언어 만큼 잘하는 것도 드물다. 따라서, C언어는 계속 사용될 것이라고 생각되며, 그 사용 영역은 바뀌지 않을 것으로 보인다. 특히, 시스템(혹은, 하드웨어) 관련 프로그래밍에서는 C언어가 아직까지는 탁월한 성능을 보여주고 있다(물론, 가끔 어셈블리어를 사용하기도 하지만).

문제는 그렇게 쉬운 C언어를 사용하면서도 제대로 구조화된 코드를 만들어내지 못한다는 것이다. C언어를 주로 사용하는 개발자들은 어떻게 코딩을 하는지는 잘 알고 있지만, 어떤 코드가 좋은 코드인지에 대한 해답은 가지고 있지 않다. 해답을 알고 있는 경우도 있지만, 그렇다고 실무에 그것을 적용하지는 않는다. 아는 것을 직접 해보지 않으면 결국 제대로 아는 것이 아니라는 것을 느끼게 될 뿐이다. 디자인 패턴도 마찬가지다. 아는 것과 그것을 구현하는 것은 다른 행위다. 아는 것을 직접 실무에서 코드로 확인해야만 알고 있다고 말할 수 있다. 모든 코드가 완벽하지 않듯이, 우리가 만든 코드도 문제는 항상 있기 마련이다.



디자인 패턴은 작은 수준에서의 변화를 다루는 기술을 제공해 준다. 큰 수준에서는 아키텍처 패턴과 같은 것이 담당하지만, 지역적인 세밀한 구현에 대해서는 디자인 패턴이 변화와 중복을 어떻게 효과적으로 다룰 수 있는지를 가르쳐준다. 사실 소프트웨어 개발은 증분(Increments)과 변경의 연속과정이다. 즉, 이런 변화를 어떤 형식으로 관리해야 효과적이고 유연한 코드를 만들 수 있는지를 알아야 한다. 변경의 시간이 올래 걸리거나, 반복된 코드가 늘어난다는 것은 구조적으로 변화를 수용하지 못하기 때문이다. 따라서, C언어를 사용하다가 변화를 수용하는 구조를 만들 필요가 있다고 생각되면, 디자인 패턴의 아이디어를 고려해보는 것이 한 가지 해결책이 될 수 있을 것이다.

사람은 위협이라고 생각되는 외부의 자극에 대해서는 이성보다는 본능에 가까운 감성으로 대응한다. 디자인 패턴이라는 것도 처음보는 사람에게는 거부감 부터 먼저 일으킬지도 모른다. 하지만, 이것을 이겨나가는 힘은 작은 반복에 숨어있다. 작은 코드를 디자인 패턴을 이용해서 만들어보고, 그 개념을 파악한 후에 나중에 실제 업무에서 그 상황을 맞닥뜨렸을 때 사용하면 된다. 크게 한번에 변화를 이루기 보다는

작은 변화를 꾸준히 하는 것이 중요하다. 어떤 힘든 일을 하게될 때, "천리길도 한 걸음부터"라는 말로 위안을 받은적이 있을 것이다. 디자인 패턴을 C언어로 구현하는 것도 마찬가지다. 천리길을 가기 위해서 지금 한 걸음을 못뗄 이유는 없다.

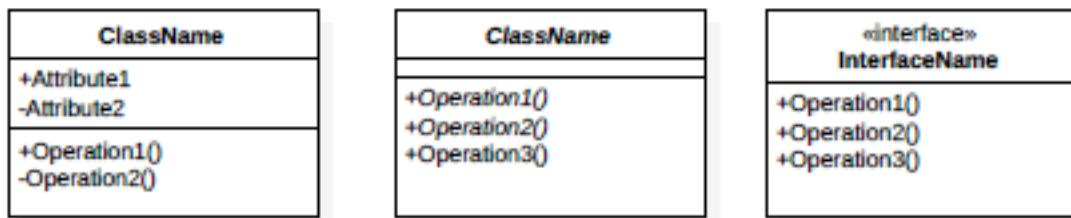
객체지향 개념이 익숙한 사람은 C언어를 사용하더라도 구조적인 코드를 만들려는 노력을 조금 더 할 것이다. 물론, C언어를 제대로 배우고 경험이 많은 사람들도 그렇게 할 것이다. 하지만, C언어만 배운 사람은 객체지향 언어를 사용해도 C언어 방식으로 코드를 작성하게 된다. 더 큰 개념의 틀을 경험하고, 작은 개념으로 한정지으면 개념을 지속시킬 수 있지만, 작은 개념에서 출발해서 갑작스럽게 만난 큰 개념에는 제대로 대응하지 못하는 것이 사실이다. 마치 작은 시냇물이 강물을 만난 것과 같이 갑작스럽게 혼란이 오는 것은 당연하다. 하지만, 그것이 역으로 거부감을 표출하는 행동으로 이어져선 안된다. 그냥 작게 한 스텝 밟 걸음을 옮기는 것으로 시작하는 것이 현명한 방법이다.

[간단한 UML(Unified Modeling Language) 익히기]

디자인 패턴을 이해하기 위해서는 간단하게라도 UML 표기법을 이해할 필요가 있다. UML은 그림으로 소프트웨어를 설명하기 위한 공통 언어의 역할을 한다. 따라서, 정확한 사용법을 이해하고 사용하면, 전 세계 누구라도 그림을 동일한 내용으로 해석할 수 있다. 마치 코딩 언어와 같이 UML도 “소통”을 목적으로 한다는 점에서는 같다고 볼 수 있다.

01. 클래스(Class)

; 클래스는 공통(Common)의 속성(Attribute)과 메소드(Method)를 가지는 같은 부류의 객체들을 분류하는 기준이다. 즉, 하나의 클래스에 속한 객체들은 클래스에 정의된 속성과 메소드를 공통으로 가진다. C언어에는 클래스 개념이 없으므로, 일단은 구조체가 변수 필드와 함수 포인터를 가지는 경우라고 정의하도록 하겠다. 물론, 객체지향 언어에서 클래스가 가져야 하는 기능(상속, 접근 권한)들을 제공해 주지 못한다. 일종의 사용자 정의 c추상 데이터 타입(User Defined Abstract Data Type)이라고 생각하면 된다.



클래스는 “이름”과 “속성”, 제공하는 “메소드(멤버 함수)”를 가질 수 있다. 클래스가 직접 객체를 생성하지 않고 클래스의 정의를 위한 타입을 제공한다면, 추상 클래스(Abstract Class)라고 한다. 추상 클래스는 “이탤릭(Italic)체”로 클래스의 이름을 표기하며, 제공하는 메소드도 이탤릭체로 된 경우에는 반드시 상속하는 클래스에서 정의해 주어야 한다. 클래스와 달리 제공하는 함수를 상속할 경우에 사용하는 것이 “인터페이스(Interface)”이다. 인터페이스는 상속하는 객체에서 직접 실체화(Realization)해 주어야 한다.

02. 오브젝트(Object)

; 클래스가 메모리내에 존재하게 될 때 오브젝트라고 한다. 즉, 오브젝트는 메모리내 생성된 클래스 타입을 따르는 변수와 같다고 볼 수 있다. 따라서, 같은 클래스에 대해서 여러 개의 오브젝트를 생성할 수 있으며, 각각은 다른 특성과 메소드를 지닐 수 있다.



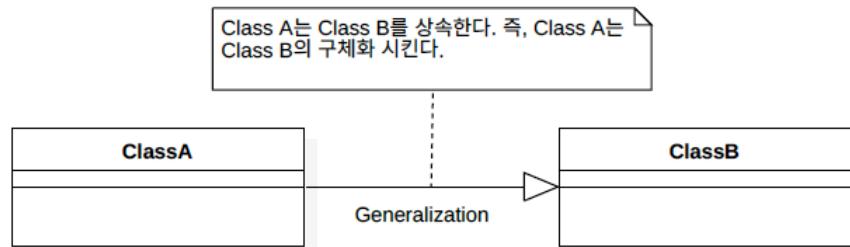
오브젝트는 밑줄 친 오브젝트 이름을 먼저 표시하고, “.” 뒤에 클래스 이름을 표시하는 방법으로 정의한다. 오브젝트는 클래스로 부터 생성 되기에, 어떤 클래스에서 나왔는지를 명시해 주어야 한다. 이미 잘 알고 있는 상황이라면 특별히 클래스 이름을 명시하지 않고, 단순히 오브젝트의 이름만 표시하는 경우도 있다. 둘다.

C언어에서는 정의된 구조체를 이용해서 생성된 사용자 변수를 오브젝트라고 하겠다. 클래스와 마찬가지로 객체지향 언어에서 제공하는 기능은 없다.

03. 관계(Relationship)

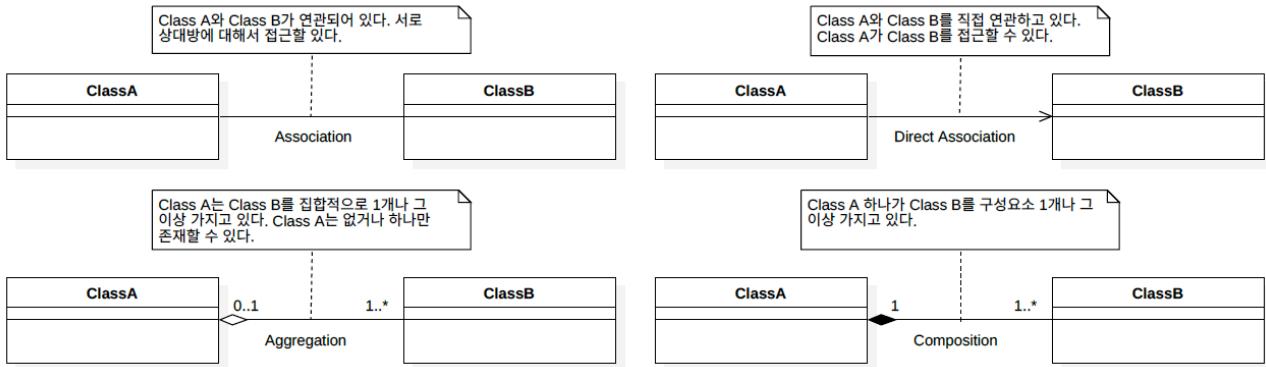
객체지향 프로그래밍에서는 절차지향 프로그래밍과는 달리 시스템을 구성하는 요소들의 “관계”로 문제를 해결한다. 따라서, 관계를 이해하는 표현하고 이해하는 것이 중요하다. 클래스 간에는 크게 아래와 같은 세 가지의 관계가 있다.

- 상속(Inheritance)** : “IS-A” - 상속을 나타내는 경우, 부모로 부터 속성(Attribute)과 메소드(Method)를 넘겨받는 경우이다. C언어는 상속이라는 개념이 없으므로, 같은 사용자 자료 구조 정도로 생각하는 것이 좋다.

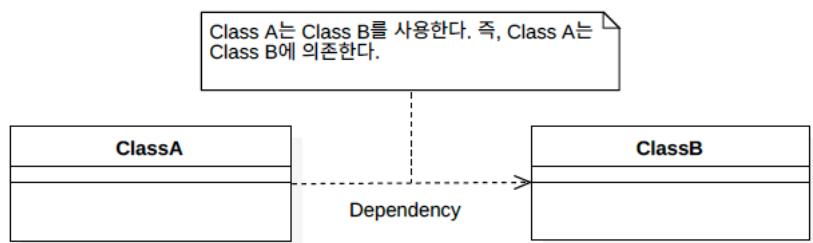


- 연관(Association)** : “HAS-A” - 객체의 내부에 다른 객체를 소유하고 있는 관계를 말한다. 즉, 다른 객체를 참조하는 필드를 가지는 경우다. 방향성을 가지는 경우는 의존 방향을 나타낸다. C언어의 경우 연관에 대해서 포인터를 사용한다고 보면 된다.

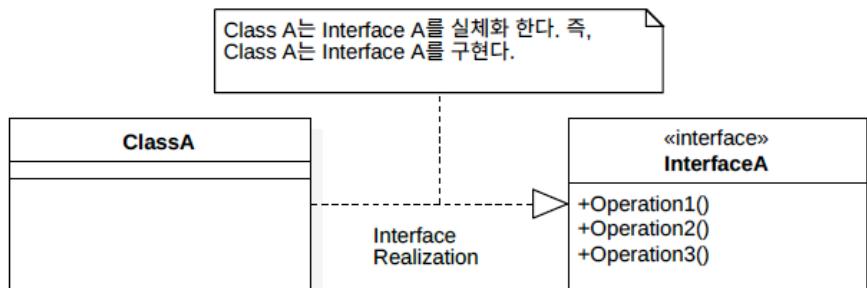
- **집합(Aggregation)** : “Whole-Part” - 전체를 가르키는 객체와 생존시간이 다른 경우(즉, 소유권을 가지는 객체가 사라져도 구성원들은 남는 경우다. 집합을 표현하기 위해서 모임을 가질 수 있는 자료구조가 필요하다. C언어의 경우 배열을 이용해서 관리한다고 생각하면 된다.
- **합성(Composition)** : “Whole-Part” - 전체를 이루는 객체와 생존시간(Lifetime)이 동일한 경우, 즉, 전체를 가르키는 객체가 사라지면 구성원들도 함께 사라지는 경우다. 합성을 구성하는 원자들을 보관할 수 있는 자료구조가 필요하다. 마찬가지로 C언어는 배열을 이용해서 관리한다고 생각하면 된다.



- **의존(Dependency)** : “USE-A” - 어떤 클래스가 다른 클래스를 참조하는 관계를 말한다. 특정 객체를 사용해서 요청을 보내는 경우, 의존을 가지는 객체를 가르키기 위해서 필드나 파라미터 등을 활용한다. C언어의 경우 의존성을 표현하기 위해서 구조체에 정의된 함수 포인터를 호출하는 것으로 생각해 볼 수 있다.



- **실체화(Realization)** : 추상적으로 정의된 인터페이스를 실체화(구현)하는 경우. 즉, “특정 클래스가 인터페이스를 구현한다”로 해석하면 된다. C언어는 인터페이스의 실체화 개념이 없으므로, 정의된 자료구조를 이용해서 실제 함수를 함수 포인터의 주소로 정해주는 과정으로 생각하면 된다.



04. 노트(Note)

; 노트는 앞의 예에서 보듯이 귀퉁이가 살짝 접힌 사각형으로 표현되며, UML에서 세부적인 설명이 필요할 때 간단히 추가해서 사용할 수 있다. 주로 역할이나 구현 방법, 목적과 같은 것을 기술하기 위해서 쓴다. 노트는 연산이 실제로 어떻게 구현 되는지 간단히 알려주기 위해서도 사용한다. 나중에 코드를 구현할 때 참고하기 위해서다.

세부적으로 상세한 UML 표기법을 여기서 다 나열할 수는 없기에, 위와 같이 코드의 구조를 표현할 수 있는 몇가지만 설명했다. 객체지향 언어의 가장 큰 특징은 “시스템을 구성하는 요소와 관계를 통해서 문제를 해결하는 것”이다. 따라서, 패턴에서 해결하는 문제들을 이해하기 위해서는 관계와 그것을 표현하는 방법을 이해해야 할 것이다.

[디자인 패턴의 분류 방법]

디자인 패턴은 목적과 범위의 2차원으로 각각 분류할 수 있다. 먼저 목적은 “생성(Creational)”, “구조(Structural)”, “행위(Behavioral)”로 나누어지며, 범위는 “클래스”와 “객체”로 구분할 수 있다.

구분	목적		
	생성	구조	행위
클래스	팩토리 메서드	어댑터(클래스)	인터프리터(해석자) 템플릿 메소드
객체	추상 팩토리 빌더 프로토타입 싱글턴(단일자)	어댑터(객체) 브리지(가교) 컴포짓 데코레이터(장식자) 파사드 플라이웨이트(경량) 프록시(대리자)	책임 연쇄 명령 반복자 중재자 메멘토 관찰자 상태 전략 방문자

C언어에서 활용하기 위해서 생성은 자료구조의 생성에 대한 것으로 보면 된다. 구조는 프로그램의 구조를 어떻게 구현할지 정하는데 활용할 수 있으며, 행위는 실행의 방법을 구현할 때 사용할 때 사용할 수 있을 것이다. 어차피 C언어에서는 클래스와 객체라는 개념이 없으므로, 앞에서 이야기 했던 것과 같이 클래스는 자료구조의 형을 표현하는 것으로 활용하고, 객체는 정의한 자료구조의 형을 실제화 시킨 사용자 변수를 다룰 때 사용하면 된다.

중요한 것은 개념을 이해하는 것이며, 활용하는 방법은 언어에 구속되지 않는다. 언어에 구속된 생각은 상상력을 제한하며, “생각은 구현하는 언어를 뛰어넘어 존재”할 수 있다. 따라서, 객체지향 언어가 아니라고 개념을 활용하지 못하는 것은 아니라는 점을 기억하기 바란다.

[주의]

앞으로 볼 패턴 들은 앞에서 정의한 구분에 따라 특정 순서로 나열한 것이 아니라, C언어 관점에서 유용 할 수 있는 것들을 우선해서 보여준다. 물론, 그렇다고 해서 반드시 여기서 나열한 순서로 중요성을 가진다는 이야기는 아니며, 참고할 수 있는 수준 정도라는 점은 밝혀둔다.

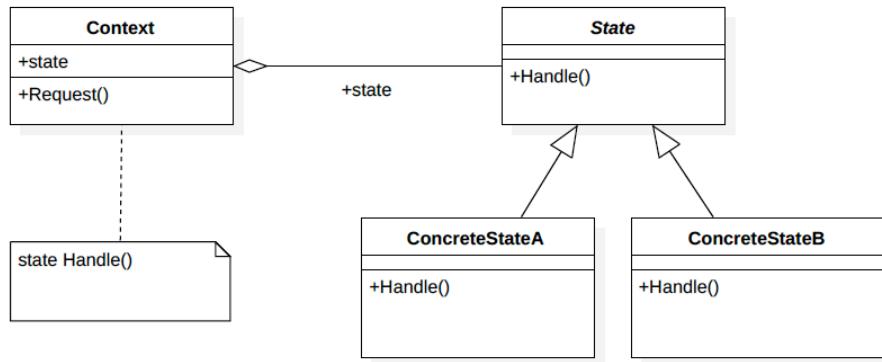
[상태 패턴(State Pattern)을 이용한 이벤트(Event)의 처리]

디자인 패턴은 자주 나오는 문제에 대한 좋은 해결책을 제시하기 위해서 만들어진 것이다. 소프트웨어의 구현은 변경이 자주 있다는 점에서, 변화를 쉽게 수용할 수 있는 방향을 제시할 필요가 있는데, 이때 각종 패턴들이 사용될 수 있다. 그리고, 그 변화가 일으킬 수 있는 코드의 수정이 최소화 될 수 있도록 도와준다. 이번에 볼 것은 유한상태(Finite State)를 가지는 기계(Machine)를 코드로 구현 것이다. 프로그램은 동작 중에 항상 특정한 상태에 있을 수 있으며, 시스템의 상태에 관계없이 받을 수 있는 입력이 동일하다고 생각할 때, 다음과 같이 상태를 기술할 수 있다.

In State A : Event X : Do Something(Action) : Next State

여기서, “Event X”는 유한상태 기계가 가질 수 있는 고정된 입력이 될 수 있으며, “State A”라는 상태에 이 입력을 받으면, 특정한 일을하고(“Do Something(Action)”), 다음 상태(“Next State”)로 이동하게 된다는 것을 표현한다. 이때, 시스템이 가지는 상태의 추가나 변경이 있을 경우, 그것을 반영하기 위한 쉬운 구현 방법이 필요하게 된다. 여기서 선택할 수 있는 방법은 각각의 “Event X”에 대해서 핸들러(Handler)들을 정의하는 방법을 사용할 수 있으며, 각각의 상태는 사용하는 측(클라이언트:Client)에서

는 동일하게 보여야 하기에, 각각의 상태가 제공하는 API의 형태는 동일해야 한다. 만약, 특정 상태에서 특정 “Event X”에 대해서 처리할 것이 없으면, 아무것도 하지 않는 기본 함수를 제공하면 된다.



시스템의 상태는 여러가지가 있으며 각각의 상태에 따라 “ConcreteStateA”와 “ConcreteStateB”가 클라이언트(“Context”)의 요청을 처리할 수 있다. 시스템의 현재 상태는 “state”를 통해서 관리 되며, 각각의 상태를 인덱스로 사용해서 배열에 저장하고 있다고 생각하면 된다. 따라서, 전달받은 요청 (“Request()”)은 각 상태가 가진 핸들러(“Handler()”)를 호출하는 것으로 처리할 수 있다.

특정 상태에서의 처리가 완료된 다음에는 다음 상태로 갈 수 있으며, 내부적인 기계의 상태와 발생한 이벤트에 따라 다음 상태가 결정될 것이다. 예를 들어, 컴파일러(Compiler)와 같은 것을 코드로 작성한다면, 프로그램의 해석(Parsing)과 같은 과정에 올바른 상태를 계속 유지하는지 검사하는데 사용할 수 있으며, 자동차와 같은 곳에서 차의 상태에 맞게 한정된 외부 이벤트에 반응하는 시스템을 만들 수도 있다. 사용자 인터페이스(UI: User Interface)와 같은 경우에도 유한상태를 가지는 기계를 정의할 수 있으며, 예를 들어, 윈도우즈와 같은 경우에는 메시지를 반복적으로 읽어와서 특정 메시지의 타입(이벤트의 타입)에 대해서 시스템이 정의된 행동을하도록 만들고 있다. 만약, 사용자가 설정한 핸들러가 없을 경우에는, 해당 이벤트는 윈도우의 기본(Default) 핸들러에서 처리하게 된다.

```

/* StatePattern.h */
#ifndef STATEPATTERN_H_
#define STATEPATTERN_H_

typedef enum {
    INIT_STATE = 0,
    READY_STATE,
    RUNNING_STATE,
    SLEEP_STATE,
    STOP_STATE,
    NONE
} STATE;

typedef struct StatePattern STATE_PATTERN;

#ifdef __cplusplus
extern "C" {
#endif

extern STATE_PATTERN *init_state_machine( void );
extern void run_state_machine( STATE_PATTERN *state );
  
```

```
#ifdef __cplusplus
}
#endif
#endif /* STATEPATTERN_H_ */
```

위의 코드에서 보듯이 현재 만들고 있는 프로그램은 상태를 한정적으로 가지고 있으며, 각 상태에 해당하는 핸들러들을 일관되게 정의하기 위해서 구조체("struct")를 이용해서 데이터와 상태의 이벤트 핸들러들을 함께 유지하고 있다. 유한 상태를 초기화 시키고, 실행시키기 위해서 두 개의 인터페이스를 제공한다.

```
/* StatePattern.c */
#include <stdio.h>
#include "StatePattern.h"

struct StatePattern {
    STATE currentState;
    void (*do_action)(STATE_PATTERN *state);
};

static void init(STATE_PATTERN *state);
static void ready(STATE_PATTERN *state);
static void running(STATE_PATTERN *state);
static void sleep(STATE_PATTERN *state);
static void stop(STATE_PATTERN *state);

static void init(STATE_PATTERN *state) {
    printf("The current state is : INIT!!!\n");
    state->currentState = READY_STATE;
    state->do_action = ready;
}

static void ready(STATE_PATTERN *state) {
    printf("The current state is : READY!!!\n");
    state->currentState = RUNNING_STATE;
    state->do_action = running;
}

static void running(STATE_PATTERN *state) {
    printf("The current state is : RUNNING!!!\n");
    state->currentState = SLEEP_STATE;
    state->do_action = sleep;
}

static void sleep(STATE_PATTERN *state) {
    printf("The current state is : SLEEP!!!\n");
    state->currentState = STOP_STATE;
    state->do_action = stop;
}

static void stop(STATE_PATTERN *state) {
    printf("The current state is : STOP!!!\n");
```

```

state->currentState = NONE;
state->do_action = init;
}

static STATE_PATTERN machine;

STATE_PATTERN *init_state_machine(void) {
    machine.currentState = INIT_STATE;
    machine.do_action = init;

    return &machine;
}

void run_state_machine(STATE_PATTERN *state) {
    do {
        state->do_action(state);
    } while (state->currentState != NONE);
}

```

위의 코드는 간단히 각각의 상태에 따른 이벤트 핸들러 들이 어떻게 정의 되었는지 보여주고 있다. 여기서는 단순히 자신의 상태를 출력하고, 다음 상태가 어디로 가야하는지와, 그 상태에서 필요한 핸들러를 등록하는 것으로 한정했다. 다음 상태에 도달했을 때는 다른 핸들러가 동일한 이벤트에 대해서 처리를 담당할 것이다. 추가적인 상태를 정의하고자 한다면, 상태 값과 핸들러를 등록하고, 어느 핸들러에서 상태를 변경할 것인지를 결정해 주어야 한다. 물론, 처리 완료 후에는 시스템이 어떤 상태로 갈 것인지도 알려주어야 한다.

유한상태 기계가 동작한다는 것을 보여주기 위해서 "run_state_machine()" 함수를 만들었다. 여기서는 모든 상태의 수행을 끝나면 반복을 빠져 나오도록 정의했다. 사용자나 시스템 외부에서의 이벤트가 없기에, 단순히 반복문을 실행하면서 유한상태 기계의 동작성을 테스트 할 뿐이다. 실제 구현에서는 이것을 응용해서 자신의 목적에 맞는 시스템을 만들면 된다.

```

/* Main.c */
#include <stdio.h>
#include <stdlib.h>

#include "StatePattern.h"

int main(void) {
    puts("State Pattern Example 01"); /* prints State Pattern Example 01 */

    STATE_PATTERN *machine = init_state_machine();
    run_state_machine( machine );

    return EXIT_SUCCESS;
}

```

"main()"함수에서는 상태를 초기화 하고, 실행 시키는("run_state_machine()") 것만 해주었다. 유한상태 기계가 실행된 이후에는 처리가 완료되지 않을 가능성도 있으며, 임베디드 시스템과 같은 경우에는 이곳으로 복귀하지 않을수도 있다. 예를 들어, 자체 스케줄러와 이벤트 전달 메커니즘을 구현한다면, 여기서 보여주는 "main()"함수는 단순히 유한상태 기계를 실행하는 역할만 해줄 뿐이다.

[결과]

```
State Pattern Example 01
The current state is : INIT!!!
The current state is : READY!!!
The current state is : RUNNING!!!
The current state is : SLEEP!!!
The current state is : STOP!!!
```

이번에는 앞에서 보여준 예제를 약간 변형해서, 각각의 상태(State)에 맞는 자료구조를 정의하고, 그 상태에서 해야할 일을 자료구조의 내부에 지정했다(do_something()). 즉, 상태를 정의하는 자료구조를 교체하면서, 상태에 맞는 일을 처리할 수 있도록 변경한 것이다.

```
#ifndef STATEMACHINE_H_
#define STATEMACHINE_H_

typedef struct state_machine STATE_MACHINE;

typedef enum {
    INIT_STATE = 0,
    READY_STATE,
    RUNNING_STATE,
    WAIT_STATE,
    STOP_STATE,
    NONE_STATE
} MACHINE_STATE;

#endif __cplusplus
extern "C" {
#endif

STATE_MACHINE *init_state_machine( void );
void run_state_machine( STATE_MACHINE *state );

#endif __cplusplus
}
#endif
#endif /* STATEMACHINE_H_ */
```

시스템이 어떤 상태를 가질지를 정의했다. 마지막 상태("NONE_STATE")는 시스템이 종료 되었다는 것을 표현하며, 이곳에서는 상태의 끝을 나타내기 위해서 사용했다. 유한상태 기계를 초기화 하고, 그것을 실행하기 위한 함수도 이곳에서 지정했다.

```
#include <stdio.h>
#include <stdlib.h>

#include "StateMachine.h"

struct state_machine {
    MACHINE_STATE state;
    void (*do_something)(STATE_MACHINE *state);
};
```

```

static void print_current_state(STATE_MACHINE *state) {
    printf("The Current Machine State : ");
    switch (state->state) {
        case INIT_STATE:
            printf("Init State!!!\n");
            state->state = READY_STATE;
            break;
        case READY_STATE:
            printf("Ready State!!!\n");
            state->state = RUNNING_STATE;
            break;
        case RUNNING_STATE:
            printf("Running State!!!\n");
            state->state = WAIT_STATE;
            break;
        case WAIT_STATE:
            printf("Wait State!!!\n");
            state->state = STOP_STATE;
            break;
        case STOP_STATE:
            printf("Stop State!!!\n");
            state->state = NONE_STATE;
            break;
        default:
            printf("Cannot Determine Current State!!!\n");
            break;
    }
}

```

각각의 상태는 자신의 상태를 나타내는 변수와 현재 상태에서 해주어야 할 일을 함수의 포인터로 가진다. 만약, 처리해야 할 이벤트의 종류가 늘어난다면, 이벤트의 종류만큼 처리 함수(Handler Function)을 추가할 수 있도록 함수 포인터를 늘리면 될 것이다. 그 함수를 호출하는 것은 외부에서 인터페이스를 이용해서 할 수 있도록 만들어주면 될 것이다. 여기서는 단순히 자신의 상태를 표시하고, 다음 상태가 무엇인지를 표시했다. 즉, 처리의 결과가 시스템의 상태 변화를 일으킬 경우에는 이렇게 상태를 변경하는 것도 추가되어야 할 부분이다.(구현을 간단하게 만들기 위해서 공통 함수를 만들고, “switch()”문을 사용했을 뿐이다.)

```

/* 계속 */
STATE_MACHINE machine_init  = { INIT_STATE, print_current_state };
STATE_MACHINE machine_ready  = { READY_STATE, print_current_state };
STATE_MACHINE machine_running = { RUNNING_STATE, print_current_state };
STATE_MACHINE machine_wait   = { WAIT_STATE, print_current_state };
STATE_MACHINE machine_stop   = { STOP_STATE, print_current_state };
STATE_MACHINE machine_none   = { NONE_STATE, print_current_state };

```

상태를 표현하는 구조체들과 자신의 상태에 취해야 할 행동(Action)을 이곳에서 초기화 시켜주었다. 이들 각각의 상태는 동일한 형(Type)을 가지기에, 같은 포인터를 이용해서 접근할 수 있다. 또한, 이벤트 핸들러들의 형식도 동일 하기에, 특정 이벤트를 핸들러를 호출해서 처리해 줄 수 있을 것이다. 앞에서 이야기 했던 것과 마찬가지로 처리해야 할 이벤트가 늘어나면, 그것에 맞춰서 핸들러를 추가하거나, 전체 이벤트를 처리하는 핸들러를 정의하고, 내부에서 각각의 이벤트에 맞는 처리로 나누어 줄 수 있을 것

이다. 즉, "print_current_state()"를 각 상태 별로 처리 핸들러를 만들고, 그 핸들러의 내부에서 이벤트의 분기를 대응해 주면 될것이다. 만약, 해당 상태에서 이벤트를 처리할 필요가 없다면, 그냥 아무것도 안하는 디폴트 핸들러로 대체 가능하다.

```
/* 계속 */
static STATE_MACHINE *do_state_machine(STATE_MACHINE *state) {
    STATE_MACHINE *newState = NULL;

    switch (state->state) {
        case INIT_STATE:
            newState = &machine_ready;
            break;
        case READY_STATE:
            newState = &machine_running;
            break;
        case RUNNING_STATE:
            newState = &machine_wait;
            break;
        case WAIT_STATE:
            newState = &machine_stop;
            break;
        case STOP_STATE:
            newState = &machine_none;
            break;
        default:
            printf("Cannot Determine Current State!!!\n");
            break;
    }
    state->do_something(state);
    return newState;
}
```

위의 코드는 상태 변화를 기술하기 위해서 만들어진 것이다. 여기서는 이전의 상태를 반영해서 다음 상태를 유추하고, 해야 할 일을 실행하는 코드를 만들어 두었다("state->do_something()"). 그리고, 상태 변화를 발생 시켰기에, 새로운 상태를 표현하는 자료구조를 복귀 값으로 돌려주었다. 다음에 이벤트가 발생하면 새로운 상태를 나타내는 자료구조가 이를 담당하게 될 것이다.

```
/* 계속 */
STATE_MACHINE *init_state_machine( void ) {
    return &machine_init;
}

void run_state_machine( STATE_MACHINE *state ) {
    STATE_MACHINE *machine;

    /* Run the state machine here */
    do {
        machine = do_state_machine( state );
    }while( machine != NULL );
}
```

위의 코드는 간단히 유한상태 기계를 초기화하고, 실행하도록 만들었다. 물론, 코드를 조정해서 무한으로 동작하게 만들수도 있지만, 이전의 코드에서 종료조건을 명시한 것을 이곳에서 검사하도록 했다. 일반적으로 유한상태 기계는 완료조건을 가지지만, 임베디드 시스템과 같은 곳에서 사용될 경우에는 전원 차단(Power Off)과 같은 경우에만 실행을 완료하는 경우가 많다. 전원 차단도 일종의 이벤트이기에 시스템을 다시 켰을 때 안정적으로 동작할 수 있는 상태를 만들고 종료해야 할 것이다.

```
#include <stdio.h>
#include <stdlib.h>

#include "StateMachine.h"

int main(void) {
    puts("State Pattern Example 02"); /* prints State Pattern Example 02 */

    STATE_MACHINE *state;
    state = init_state_machine();
    run_state_machine( state );

    return EXIT_SUCCESS;
}
```

위의 코드는 단순히 유한상태 기계를 초기화하고, 실행하도록 명령만 내렸을 뿐이다. 임베디드 시스템과 같은 경우에는 유한상태 기계가 동작을 마친 이후에 이곳으로 복귀하지 않을 수도 있다.

[결과]

```
State Pattern Example 02
The Current Machine State : Init State!!!
The Current Machine State : Ready State!!!
The Current Machine State : Running State!!!
The Current Machine State : Wait State!!!
The Current Machine State : Stop State!!!
Cannot Determine Current State!!!
The Current Machine State : Cannot Determine Current State!!!
```

위의 출력 결과에서 보듯이 각각의 상태를 한 번씩 실행하고 유한상태 기계는 종료한 것을 볼 수 있다. 만약, 새로운 상태를 정의 하고자 한다면, 동일한 자료구조를 이용해서 상태를 표현하는 자료구조를 선언하고, 이벤트 핸들러들을 정의한 후에, 어떤 상태에서 어떤 이벤트가 발생했을 때, 해당 상태의 자료구조체를 사용할 것인가 정해주어야 할 것이다. 마찬가지로 추가되는 상태의 자료구조에서도 이벤트 핸들러의 처리 결과가 시스템의 상태 변화를 일으킬 때, 어떤 상태로 갈지를 결정해 주어야 한다.

다음의 코드에서는 각각의 상태마다 전달되는 이벤트에 맞게 핸들러를 정의해서, 이벤트와 그에 관련된 메시지를 처리하는 것을 간단히 보여주겠다. 즉, 시스템의 상태에 맞는 핸들러들을 각각 정의하고, 그 상태에서 전달되는 이벤트에 각각 나누어서 어떻게 처리하는지를 구현했다. 이벤트 처리 핸들러들은 동일한 이벤트에 대해서 반응하며, 각각을 자신의 상태에 맞춰서 처리할 있도록 했다.

```
#ifndef STATEMACHINE_H_
#define STATEMACHINE_H_

typedef struct StateMachine STATE_MACHINE;
```

```
typedef enum {
    STATE_INIT = 0,
    STATE_READY,
    STATE_RUNNING,
    STATE_WAITING,
    STATE_STOPPED,
    STATE_NONE
} STATE;
```

```
typedef enum {
    EVENT_PRINT = 0,
    EVENT_NEXT,
    EVENT_PREVIOUS,
    EVENT_EXIT,
    EVENT_NONE
} EVENT;
```

```
typedef struct message {
    char *msg;
} MESSAGE;
```

```
#ifdef __cplusplus
extern "C" {
#endif
```

```
extern void init_state_machine( void );
extern void run_state_machine( void );
```

```
#ifdef __cplusplus
}
#endif
#endif /* STATEMACHINE_H_ */
```

헤더 파일에는 시스템의 상태를 만들기 위한 자료구조와("STATE_MACHINE"), 시스템의 상태("STATE"), 각 상태마다 처리해야 할 이벤트("EVENT")를 정의 했다. 그리고, 이벤트가 발생했을 때 전달받을 수 있는 메시지("MESSAGE")도 정의했다. 이번 예제는 시스템의 상태를 초기화하고, 유한상태 기계를 실행하기 위한 두 개의 인터페이스만 외부로 노출한다.

```
#include <stdio.h>
#include "StateMachine.h"

typedef void (*HANDLER)( char *state, MESSAGE *msg );

struct StateMachine {
    char *name;
    HANDLER handler[ EVENT_NONE ];
};

static void event_print_message( char *state, MESSAGE *msg ) {
    printf( "%s : Print Handler Called : %s!!!\n", state, msg->msg );
```

```

}

static void event_next_message( char *state, MESSAGE *msg ) {
    printf( "%s : Next Handler Called : %s!!!\n", state, msg->msg );
}

static void event_previous_message( char *state, MESSAGE *msg ) {
    printf( "%s : Previous Handler Called : %s!!!\n", state, msg->msg );
}

static void event_exit_message( char *state, MESSAGE *msg ) {
    printf( "%s : Exit Handler Called : %s!!!\n", state, msg->msg );
}

```

각각의 상태는 상태의 이름("name")과 각각의 이벤트에 대한 나누어진 핸들러 들의 배열("handler[]")로 구성되어 있다. 즉, 시스템의 특정 상태에서 전달되는 이벤트를 처리할 수 있는 핸들러 들을 동일하게 정의한다. 따라서, 각각의 상태에서는 동일한 이벤트를 받아서, 자신이 원하는데로 처리할 수 있는 기회가 생기게 된다. 여기서는, 각각의 상태 별로 따로 핸들러 들의 배열을 정의한 것이 아니라, 간단한 예를 보여주기 위해서, 동일한 함수들을 다른 메시지로 호출 할 수 있도록 정의했다.

```

/* 계속 */
STATE_MACHINE init = { "Init", { event_print_message, event_next_message,
                                event_previous_message, event_exit_message } };
STATE_MACHINE ready = { "Ready", { event_print_message, event_next_message,
                                event_previous_message, event_exit_message } };
STATE_MACHINE running = { "Running", { event_print_message, event_next_message,
                                         event_previous_message, event_exit_message } };
STATE_MACHINE waiting = { "Waiting", { event_print_message, event_next_message,
                                         event_previous_message, event_exit_message } };
STATE_MACHINE stopped = { "Stopped", { event_print_message, event_next_message,
                                         event_previous_message, event_exit_message } };

STATE_MACHINE *state_machine[ STATE_NONE ] = { &init, &ready, &running, &waiting,
                                              &stopped };
static STATE currentState = STATE_NONE;

```

앞의 코드는 유한상태 기계를 나타내는 각각의 상태들을 구성했다. 상태의 이름과 그 상태에서 이벤트에 맞게 호출되어야 할 함수들의 포인터를 전부 나열했다. 즉, 상태를 이용해서 현재 상태에 맞는 핸들러를 찾고, 그렇게해서 찾은 핸들러에서 이벤트에 해당하는 함수를 호출하게 되는 것이다. 현재 상태를 기록하기 위해서 내부 변수로 "currentState"를 사용했다.

```

/* 계속 */
MESSAGE init_msg = { "Print Message" };
MESSAGE next_msg = { "Next Message" };
MESSAGE previous_msg = { "Previous Message" };
MESSAGE exit_msg = { "Exit Message" };

static void handle_state_machine( EVENT event, MESSAGE *msg ) {
    STATE_MACHINE *state;

    state = state_machine[ currentState ];

```

```

        state->handler[ event ]( state->name, msg );
    }

void init_state_machine( void ) {
    currentState = STATE_INIT;
}

void run_state_machine( void ) {
    while( currentState != STATE_NONE ) {
        handle_state_machine( EVENT_PRINT, &init_msg );
        handle_state_machine( EVENT_NEXT, &next_msg );
        handle_state_machine( EVENT_PREVIOUS, &previous_msg );
        handle_state_machine( EVENT_EXIT, &exit_msg );
        currentState++;
    }
}

```

각각의 상태 별로 자신의 핸들러가 호출 되었다는 것을 알기 위해서, 이벤트의 핸들러에서 정해진 메시지를 출력하는 것으로 만들었다. 나중에 핸들러를 구현할 때 원하는 방법으로 이벤트 핸들러 들을 적절히 바꾸면 될 것이다. 실제 시스템의 이벤트를 상태에 맞게 처리하는 함수는 "handle_state_machine()"이 담당하고 있다. 이 함수는 현재의 시스템 상태에 맞게 핸들러를 찾고, 핸들러가 가지고 있는 이벤트 핸들러들을 배열의 인덱스 방법으로 찾도록 구현했다. 즉, 이벤트 타입에 따라, 처리해주는 함수들을 배열로 배치했다. 호출되는 함수에는 전달받은 메시지를 넘겨주어, 해당 이벤트에서 처리해야 할 내용을 명확히 할 수 있다.

```

#include <stdio.h>
#include <stdlib.h>
#include "StateMachine.h"

int main(void) {
    puts("State Pattern Example 03");
    init_state_machine();
    run_state_machine();
    return EXIT_SUCCESS;
}

```

"main()" 함수에서는 단순히 유한상태 기계를 초기화하고, 이벤트와 메시지를 전달해서 전체 유한상태 기계가 제대로 동작 하는지를 검증하고 있다. 코드를 조금만 수정하면 반복문을 사용해서 이벤트를 받아서 시스템의 상태에 맞게 핸들러들을 호출하는 부분을 넣을 수 있을 것이다. 유한상태 기계를 이용하는 것은 입력이 항상 한정된 범우를 가지며, 어떤 상태에서도 그런 이벤트가 발생할 수 있다는 가정이 있다. 물론, 발생한 이벤트를 처리 하느냐 그렇지 않느냐는 선택하기 나름이다.

[결과]

```

State Pattern Example 03
Init : Print Handler Called : Print Message!!!
Init : Next Handler Called : Next Message!!!
Init : Previous Handler Called : Previous Message!!!
Init : Exit Handler Called : Exit Message!!!
Ready : Print Handler Called : Print Message!!!
Ready : Next Handler Called : Next Message!!!
Ready : Previous Handler Called : Previous Message!!!

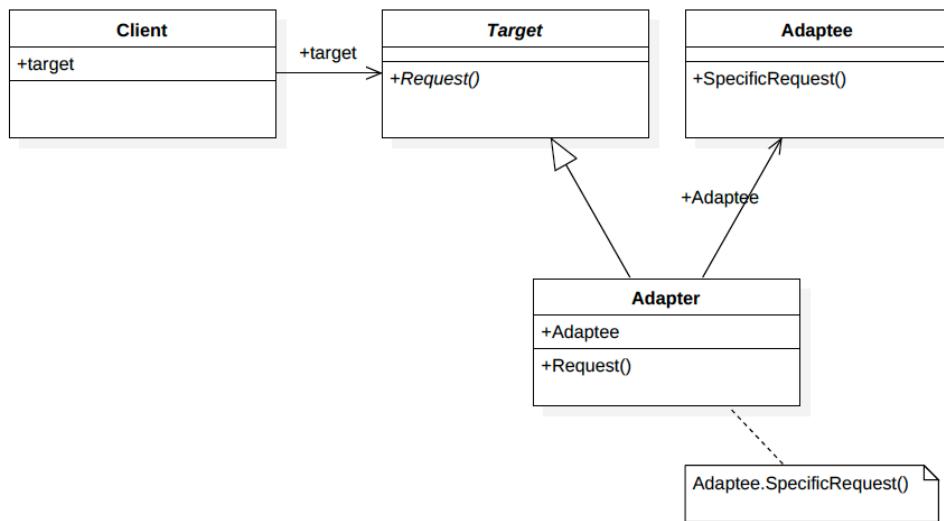
```

Ready : Exit Handler Called : Exit Message!!!
 Running : Print Handler Called : Print Message!!!
 Running : Next Handler Called : Next Message!!!
 Running : Previous Handler Called : Previous Message!!!
 Running : Exit Handler Called : Exit Message!!!
 Waiting : Print Handler Called : Print Message!!!
 Waiting : Next Handler Called : Next Message!!!
 Waiting : Previous Handler Called : Previous Message!!!
 Waiting : Exit Handler Called : Exit Message!!!
 Stopped : Print Handler Called : Print Message!!!
 Stopped : Next Handler Called : Next Message!!!
 Stopped : Previous Handler Called : Previous Message!!!
 Stopped : Exit Handler Called : Exit Message!!!

위의 출력 내용에서 보듯이 인위적으로 각각의 상태에서 정의된 이벤트 핸들러들을 전부 호출됨을 확인할 수 있다. 새로운 상태를 추가하는 것은 새로운 상태를 지정하고, 관련된 이벤트 핸들러들을 추가하면 될 것이다. 만약, 자신에게 해당하지 않는 이벤트가 발생한다면, 그냥 무시해도 될 것이다. 구현에서 보듯이 새로운 상태를 추가하는 것도, 새로운 이벤트를 추가하는 것도 어렵지 않다는 것을 알 수 있을 것이다. 즉, 변경에 대해서 좀 더 유연한 구조를 유지할 수 있도록 만들었다.

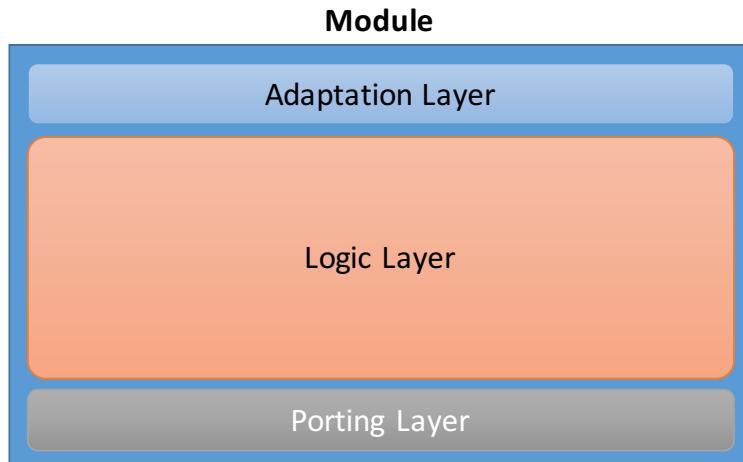
[어댑터 패턴(Adapter)을 이용한 차이의 극복]

어댑터는 한 모듈과 다른 모듈을 붙여야 할 경우, 서로 간에 사용하는 인터페이스의 차이를 극복하기 위해서 사용할 수 있다. 예를 들어, 의존관계에 있는 한 모듈에서 제공하는 인터페이스가 우리가 필요한 일부만을 제공할 경우, 이를 변경해서 우리가 원하는 인터페이스로 만들어 준다. “3rd Party”와 같은 곳에서 제공하는 API가 우리가 필요한 것과는 다른 경우, 제공받는 코드를 변경하는 것은 불가능하다. 오픈 소스라면 그나마도 코드를 볼 수 있지만, 상업적으로 사용 되는 것이라면, 코드를 직접 가져다 사용할 수 없다.



클라이언트는 자신의 요청을 누가 처리할지 알고 있으며(“Target”), 실제 처리를 맡고 있는 “Adaptee”에 요청을 전달해서 처리를 의뢰하는 것은 “Adapter”的 역할이다. 따라서, “Adapter”를 클라이언트가 안다면, 자신의 요청이 어떻게 처리될지 신경쓰지 않아도 된다. “Adapter”는 클라이언트가 알고 있는 “Target”的 역할만 충실히 수행(“Request()”)할 수 있으면 되는 것이다.

데이터 베이스(DataBase)와 같은 경우, 서로 다른 API를 제공하지만, 기능적으로는 유사한 경우 다른 것으로 교체하기 위해서도 어댑터 부분은 자주 필요하다. 모듈을 만들 때도 어댑터를 사용할 수 있는데, 이 경우에는 한 모듈이 외부에 의존적인 부분들을 특정 인터페이스의 집합으로 정의해서 계층을 분리하고, 이 계층을 정말 의존적인 부분을 위한 포팅(Porting) 계층으로 사용하는 것이다. 이것을 기본으로 가져간다면, 외부로 공개 되는 인터페이스 계층, 모듈 자체에서 처리하는 기능을 구현하는 계층, 모듈이 외부에 의존하는 인터페이스 계층 등으로 세 개의 계층으로 한 모듈을 나눌 수 있을 것이다.



계층을 정의하는 것은 성능상의 불이익이 있지만, 그것으로 인해서 더 유연하게 변화에 대응할 수 있다는 점과, 코드의 재사용 가능성을 높인다는 점에서는 충분한 이점을 제공한다. 따라서, 기본적으로 모듈을 설계하는 입장에서는 적어도 세 개의 계층을 가져가는 것이 좋다. 물론, 계층이라고 말하고 있지만, 실질적으로는 2개의 계층일 수도 있다. 즉, 의존하는 부분을 단순히 제공받을 인터페이스 수준에서만 정의할 수 있을 것이며, 제공해야 할 인터페이스 역시 단순히 헤더 파일이의 형태로 묶어서 생각해 볼 수 있다.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    char *test_string = "0123456789";
    char result_string_without_zero[100];
    char result_string_with_zero[100];

    memset( result_string_without_zero, 0, sizeof( char ) );
    memset( result_string_with_zero, 0, sizeof( char ) );
    printf("The length of string : %d\n", strlen( test_string ));

    strncpy( result_string_without_zero, test_string, strlen( test_string ) );
    strncpy( result_string_with_zero, test_string, strlen( test_string ) + 1 );
    printf( "The result string : %s\n", result_string_without_zero );
    printf( "The result string : %s\n", result_string_with_zero );

    return EXIT_SUCCESS;
}
  
```

[결과]

The length of string : 10

The result string : 0123456789?v

The result string : 0123456789

위의 코드는 간단히 "strlen()" 함수의 기능을 보기위해서 작성했다. 즉, "strlen()" 함수는 string으로 정의된 문자열을 시작에서 끝까지 길이를 알아오는데 사용한다. 한 가지 문제는 "strlen()" 함수가 문자열의 마지막을 나타내는 "\0"(Terminating Null)을 길이에 포함하지 않는다는 것이다. 따라서, "strlen()" 함수를 사용해서 정확한 문자열을 복사하기 위해서는 문자열의 길이에 "1"만큼 더 해주어야 한다.

어댑터를 간단히 설명하기 위해서, 위와 같은 경우 우리가 원하는 기능을 하는 함수를 기준의 "strlen()" 함수를 이용해서 어떻게 만드는지 보여줄 것이다. 물론, 너무 간단하지만 시작으로서는 충분한 예제라고 생각한다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

unsigned int mystrlen( char *str ) {
    unsigned int str_length = strlen( str ) + 1;
    return str_length;
}

int main(void)
{
    char *test_string = "0123456789";
    char result_string_without_zero[100];
    char result_string_with_zero[100];

    memset( result_string_without_zero, 0, sizeof( char ) );
    memset( result_string_with_zero, 0, sizeof( char ) );
    printf("The length of string : %d\n", strlen( test_string ));

    strncpy( result_string_without_zero, test_string, strlen( test_string ) );
    strncpy( result_string_with_zero, test_string, strlen( test_string ) + 1 );
    printf( "The result string : %s\n", result_string_without_zero );
    printf( "The result string : %s\n", result_string_with_zero );

    printf( "\nUsing Adapter Pattern!!!\n");
    strncpy( result_string_without_zero, test_string, mystrlen( test_string ) );
    strncpy( result_string_with_zero, test_string, mystrlen( test_string ) + 1 );
    printf( "The result string : %s\n", result_string_without_zero );
    printf( "The result string : %s\n", result_string_with_zero );

    return EXIT_SUCCESS;
}
```

위의 코드에서 보듯이, "strlen()" 함수가 가지는 취약점을 다른 함수("mystrlen()")를 이용해서 간단히 원하는 목적을 수행하도록 변경했다. 만약, 제공되는 함수들이 원하는 목적과 차이가 날 경우, 그것을 직접 이용하기보다는 새로운 API계층을 만들어 처리하는 것이 도움이 된다. 새로운 API계층을 정의하고 그것에 의존적으로 코드를 만들면, 새로운 라이브러리를 사용하거나 새로운 시스템으로 포팅하는 노력을 한 곳에만 집중할 수 있다.

위에서 만든 코드를 계층이라는 개념을 적용해서 다른 파일로 분리해 보았다. 추가적으로 문자열을 복사하는 함수도 간단히 원래의 것을 감싸는(Wrapping) 수준에서 구현해 보았다. "strcpy()"라는 함수도 "\0"을 만날 때까지 복사하는 기능이 있어 사용하는데 주의해야 하는 함수다.

```
/* MyString.h */
#ifndef MYSTRING_H_
#define MYSTRING_H_

#ifdef __cplusplus
extern "C" {
#endif

extern unsigned int mystrlen( const char *str );
extern char *mystrcpy( char *destination,const char* source);

#ifdef __cplusplus
}
#endif
#endif /* MYSTRING_H_ */

/* MyString.c */
#include <string.h>

unsigned int mystrlen(const char *str) {
    unsigned int length = strlen(str) + 1;

    if (str == NULL) {
        return 0;
    }
    return length;
}

char *mystrcpy(char *destination, const char* source) {
    return strncpy(destination, source, strlen(source) + 1);
}

/* main.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "MyString.h"

int main(void) {
    char *test_string = "0123456789";
    char result_string[100];

    printf("\nUsing Adapter Pattern Example 02!!!\n");
    printf("The string Length : %d\n", mystrlen(test_string));
    printf("The Copied string : %s\n", mystrcpy(result_string, test_string));
    return EXIT_SUCCESS;
}
```

```
}
```

[결과]

Using Adapter Pattern Example 02!!!

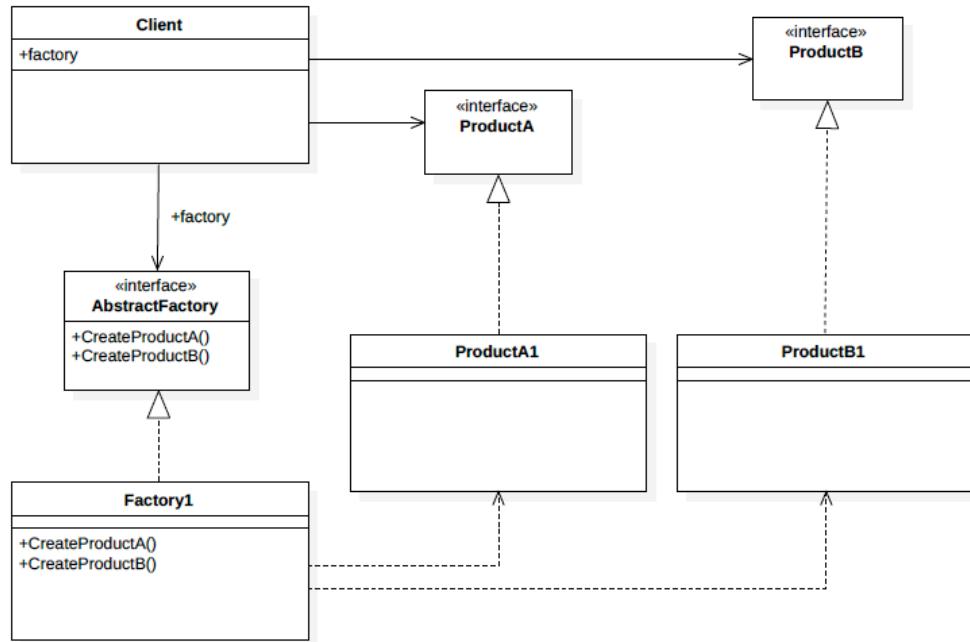
The string Length : 11

The Copied string : 0123456789

위의 결과에서 확인할 수 있듯이, 문자열을 구해서 문자열을 복제하거나 다루는데 "+1"을 해주눈 것과 "\0"에 둔감한 코드를 만들 수 있도록 해 주었다. 이처럼 어댑터는 사용자가 원하는 API의 차이를 극복하기 위해서 사용할 수 있는 패턴이며, 구체적인 구현과 분리를 통해서 좀더 변화에 융통성 있는 코드를 만들기 위해서 사용된다. 이것도 일종의 계층화를 통해서 얻을 수 있는 장점이다.

[추상 팩토리 패턴(Abstract Factory Pattern)을 이용한 다양한 제품 만들기]

추상 팩토리는 만들려고 하는 제품별로 특화된 자료구조와 인터페이스를 만들어, 사용하는 측에서는 어떤 제품을 사용 중인지 감추도록 하기 위해서 사용한다. 즉, 사용자 측은 어떤 제품을 사용하는지 모르지만 특정 인터페이스를 통해서 제품을 생산할 수 있다는 것은 알며, 자신이 알고 있는 제품 코드에 맞는 구체적인 구현을 얻도록 제품 생산을 요청하게 된다.



팩토리는 클라이언트의 요청을 받아서 클라이언트가 원하는 프로덕트를 생산하기만 하면된다. 이때, 팩토리 자체도 클라이언트와 계약 사항("AbstractFactory")를 만족시켜야 하며, 생산하는 프로덕트는 클라이언트가 요구하는 사항을 만족시킬 수 있을 정도면 된다. 따라서, 클라이언트는 실제 생산이 진행되는 팩토리와 생산된 제품인 프로덕트가 무엇인지에 상관없이 동일한 일을 수행할 수 있게 된다.

팩토리에서는 요청된 제품의 코드 별로 구체적인 제품의 생산을 담당하게 된다. 따라서, 미리 생산될 제품은 정의되어 있으며, 필요하다면 새로운 제품을 동일한 인터페이스를 따르도록 추가해 줄 수도 있다. 사용자측에서는 단순히 팩토리에 대한 인터페이스와 제품에 대한 사용법만 알 뿐이다.

실제 코드에서는 다양한 제품을 하나의 프로그램으로 대응하게 만들어야 할 경우가 있으며, 이때 주로 사용하는 방법이 제품의 코드에 따라 다르게 처리하기 위해서 "if()"나 "switch()"같은 것을 사용한다. 하지만, 이런 식으로 기능이나 제품의 종류를 확장 및 파생하게 되면, 코드 전체에 "if()"나 "switch()"가 퍼져나가게 된다. 이렇게 전체 코드에 분포하는 다양한 제품에 관련된 코드를 다루는 것은 쉬운 일이 아니

며, 중복된 코드가 작성될 가능성도 높다. 따라서, 이런 코드들을 한번에 묶어서 관리할 수 있는 방법과, 확장을 쉽게 할 수 있는 방법이 요구된다. 이때 추상 팩토리 패턴을 활용할 수 있다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef enum {
    PRODUCT_TYPE_A = 0,
    PRODUCT_TYPE_B,
    PRODUCT_TYPE_C,
    PRODUCT_TYPE_NONE
} PRODUCT_TYPE;

#define MAX_FACTORY_NAME 100
#define MAX_PRODUCT_NAME 100

typedef struct product PRODUCT;

struct product {
    char *factory_name;
    char name[ MAX_PRODUCT_NAME ];
    PRODUCT_TYPE type;
    unsigned int price;
    void (*do_something)( const PRODUCT *product );
};

static void print_product_information( const PRODUCT *product ) {
    printf( "=====\\n" );
    printf( "The Factory Name : %s\\n", product->factory_name );
    printf( "The Product Name : %s\\n", product->name );
    printf( "The Product Type : %d\\n", product->type );
    printf( "The Product Price: %d\\n", product->price );
    printf( "=====\\n" );
}
```

제품은 자신을 생산한 팩토리에 대한 정보를 알기 위해서 "factory_name"을 가지고 있다. 제품 자체의 이름도 "name"을 통해서 알 수 있으며, 제품의 타입과 가격 등의 정보를 가질 수 있다("type", "price"). 제품 자체가 해야할 일은 "do_something()" 함수에서 처리한다. 여기서는 단순히 제품의 정보를 보여주는 것으로 한정했다("print_product_information()").

```
typedef struct abstract_factory ABSTRACT_FACTORY;

struct abstract_factory {
    char *name;
    PRODUCT *(*make_product)( const ABSTRACT_FACTORY *factory, const
        PRODUCT_TYPE type, const unsigned int price );
};

PRODUCT product_none = { "Factory Name None", "Product Name None",
    PRODUCT_TYPE_NONE, 0, print_product_information };
```

추상 팩토리는 팩토리의 이름과 제품을 만들기 위한 인터페이스를 제공한다. 물론, 이런 부분을 다른 인터페이스를 정의해서 숨길 수 있으나, 여기서는 구현의 편의를 위해서 외부로 공개하는 것으로 했다. 지원하지 않는 타입의 제품도 "NULL"로 처리하지 않기 위해서 "product_none"을 정의했다. 나중에 제품의 코드가 없는 생산 요청에 대해서도 오류 확인을 거치지 않는다.

```
static PRODUCT *make_product(const ABSTRACT_FACTORY *factory,
                           const PRODUCT_TYPE type, const unsigned int price) {
    PRODUCT *product;

    if ((product = (PRODUCT *) malloc(sizeof(PRODUCT))) == NULL) {
        printf("Cannot make a product for type A!!!\n");
        return &product_none; /* Do not make people to check NULL return value */
    }

    product->factory_name = factory->name;
    switch (type) {
        case PRODUCT_TYPE_A:
            strncpy(product->name, "Product Name A", strlen("Product Name B") + 1);
            break;
        case PRODUCT_TYPE_B:
            strncpy(product->name, "Product Name B", strlen("Product Name B") + 1);
            break;
        case PRODUCT_TYPE_C:
            strncpy(product->name, "Product Name C", strlen("Product Name C") + 1);
            break;
        default:
            return &product_none;
            break;
    }
    product->type = type;
    product->price = price;
    product->do_something = print_product_information;
    return product;
}
```

```
ABSTRACT_FACTORY factory_A = { "Abstract Factory A", make_product };
ABSTRACT_FACTORY factory_B = { "Abstract Factory B", make_product };
ABSTRACT_FACTORY factory_C = { "Abstract Factory C", make_product };
```

제품을 만드는 것은 단순히 메모리를 할당받고, 제품의 정보를 설정된 값으로 정해주는 것으로 했다. 만약, 제품을 생산할 수 없거나 지원하는 타입이 아니라면, 기본으로 설정된 "product_none"의 포인터를 돌려주는 것으로 만들었다. 사용자 측에서는 "NULL"을 처리할 필요가 없다.

제품에 대한 생산을 위해서 제품의 타입에 따라 한번은 "switch()"문을 사용해야 한다. 하지만, 나중에 사용자 측의 코드에서는 더 이상 "switch()"에 의존해서 각각의 제품 타입에 맞게 처리하는 것은 하지 않아도 된다. 즉, 전체 코드에 제품의 타입에 맞는 구현은 사라지게 된다.

각각의 제품에 대해서 각각의 팩토리를 정했다. 실제 사용시에는 제품의 타입 만으로 팩토리를 접근할 수 있도록 바꾸게 되면, 어떤 팩토리를 사용하는지 알지 못하고 자신이 원하는 타입의 제품을 생산할 수

있도록 할 수 있다. 만약, 하나의 팩토리만 사용하고자 한다면, 해당 팩토리에서 제품의 타입별로 각각의 제품을 생산하도록 만들수도 있다. 여기서는 제품 별로 팩토리를 분리해서 사용하도록 구현했다.

```
int main(void) {
    puts("Abstract Factory Example 01");
    PRODUCT *product;

    product = factory_A.make_product( &factory_A, PRODUCT_TYPE_A, 100 );
    product->do_something( product );
    product = factory_B.make_product( &factory_B, PRODUCT_TYPE_B, 10 );
    product->do_something( product );
    product = factory_C.make_product( &factory_C, PRODUCT_TYPE_C, 150 );
    product->do_something( product );
    product = factory_A.make_product( &factory_A, PRODUCT_TYPE_NONE, 0 );
    product->do_something( product );

    return EXIT_SUCCESS;
}
```

예제 코드는 특정 팩토리에서 생산된 제품을 사용하는 것을 보여준다. 각각의 팩토리에서 생산된 제품들은 동일한 포인터에 의해서 접근할 수 있으며, 공통된 인터페이스를 가지고 있기에, 그것을 이용해서 제품의 정보를 표시하도록 만들었다. 즉, 사용자의 코드는 어떤 제품을 사용하는지 모르며, 일관된 인터페이스를 통해서 생산된 제품을 사용할 수 있게 되는 것이다. 따라서, 제품 타입 별로 분리되어야 했던 처리를 더이상 보지 않아도 된다.

[결과]

Abstract Factory Example 01

```
=====
The Factory Name : Abstract Factory A
```

```
The Product Name : Product Name A
```

```
The Product Type : 0
```

```
The Product Price: 100
```

```
=====
The Factory Name : Abstract Factory B
```

```
The Product Name : Product Name B
```

```
The Product Type : 1
```

```
The Product Price: 10
```

```
=====
The Factory Name : Abstract Factory C
```

```
The Product Name : Product Name C
```

```
The Product Type : 2
```

```
The Product Price: 150
```

```
=====
The Factory Name : Factory Name None
```

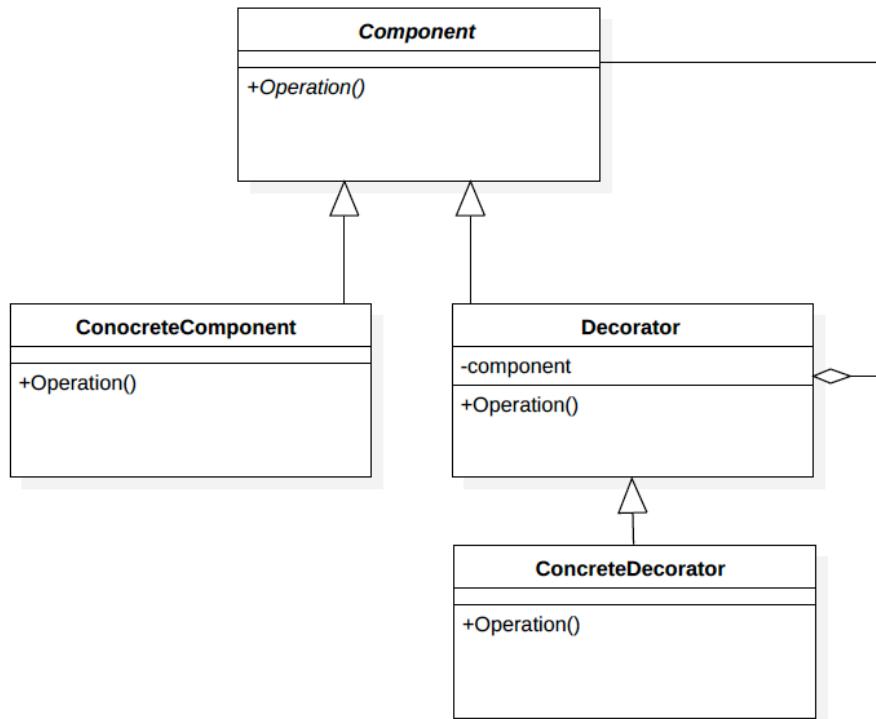
```
The Product Name : Product Name None
```

```
The Product Type : 3
```

```
The Product Price: 0
```

[장식자 패턴(Decorator Pattern)을 이용한 꾸미기]

장식자(Decorator) 패턴은 말 그대로 기존의 데이터를 유지하면서 추가적인 처리를 덧붙이기 위해서 사용하는 패턴이다. 예를 들어, 윈도우즈 응용프로그램에서 화면에 나타나는 데이터를 유지하면서, 폰트를 변경하거나, 윈도우의 창틀을 변경하는 것, 각종 메뉴바를 더하는 것과 같은데서 활용해 볼 수 있을 것이다. 프로토콜과 같은 경우, 각각의 계층을 데이터가 통과하면서 헤더(Header)와 테일(Tail) 등에 데이터를 추가하는 곳에서도 사용할 수 있다. 만약, 새로운 프로토콜이 계층 사이에 추가되거나, 혹은 윈도우의 새로운 속성을 주고자 한다면, 장식을 담당할 부분을 간단히 추가하면 된다. 여기서는 이를 구현할 수 있는 간단한 아이디어 정도를 소개하는 정도로 보여주도록 하겠다.



장식자는 여러 개의 컴포넌트가 연결된 구조를 가지며, 각각은 동일한 인터페이스를 정의한다. 즉, 연결 구조를 따라가면서 정의된 연산("Operation()")을 실행할 수 있게 되는 것이다. 얼마나 많이 연결될지 알 수 없기에, 배열을 사용해선 안되며 포인터를 이용한 연결리스트를 사용하는 것으로 구현 가능하다. 컴포넌트는 동적으로 추가될 수 있기에 필요한 연산이 있을 경우 컴포넌트 구조로 생성해서 연결리스트에 추가하면 된다.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Message {
    char message[100];
} MESSAGE;

typedef struct Decorator DECORATOR;
struct Decorator {
    char *decorator_name;
    DECORATOR *next_decorator;
    void (*do_something)(MESSAGE *msg);
}
  
```

```

};

void do_something_method(MESSAGE *msg) {
    printf("Message : %s\n", msg->message);
    strcat(msg->message, "More!!!");
    return;
}

DECORATOR myThirdDecorator = { "Third Decorator", NULL, do_something_method };
DECORATOR mySecondDecorator = { "Second Decorator", &myThirdDecorator,
                                do_something_method };
DECORATOR myFirstDecorator = { "First Decorator", &mySecondDecorator,
                               do_something_method };

```

각각의 데코레이터들은 하나의 처리 계층과 같은 역할을 한다. 여기서는 간단히 구조체("struct")를 이용해서 이런 계층을 정의해 주었다. 각각이 자신의 이름과 다음 데코레이터에 대한 포인터를 유지하며, 자신에게 데이터 처리의 기회가 왔을 때, 이를 담당할 함수를 정의해 주었다. 여기서는 단순히 각각의 계층화 된 데코레이터에서 문자열을 뒤에 추가하도록 했다.

```

void do_decorator(DECORATOR *deco, MESSAGE *msg) {
    do {
        deco->do_something( msg );
        deco = deco->next_decorator;
    } while(deco != NULL);
}

```

위의 코드는 각각의 데코레이터들을 차례로 방문하면서, 메시지를 처리하도록 만든 것이다. 단순화 시킨 면은 있지만, 위에서 만든 계층구조에 새로운 구조체를 정의해서 연결시켜주면, 기존의 데이터 처리에 변화를 줄 수 있다는 점을 알 수 있다. 즉, 기존의 코드에 대한 변경을 최소화하면서, 추가적인 기능을 더할 수 있다는 것이다.

```

int main(void) {
    puts("Decorator Example 01");
    MESSAGE msg = { "Let's enjoy!!!" };

    do_decorator(&myFirstDecorator, &msg);
    return EXIT_SUCCESS;
}

```

위의 코드는 제공받은 API와 메시지의 처리를 담당한 자료구조에 대한 함수 호출로 이뤄져 있다. 즉, 내부에서 어떤 처리가 일어날지는 모르며, 단순히 외부에 드러난 API와 자료구조만을 사용했을 뿐이다. 추가되는 자료구조와 데코레이터들에 대해서는 호출하는 측에서 알 필요가 없다. 위의 프로그램을 실행한 결과는 아래와 같다. 여기서는 처리되는 순서에 따라 문자열이 길어짐을 확인할 수 있을 것이다. 순서를 자유롭게 정의되기 위해서는 좀 더 일반화된 메시지에 대한 정보를 정의해야 하겠지만, 이 예제에서는 순서를 이용해서 문자열을 다루는 것만 보여주었다.

[결과]

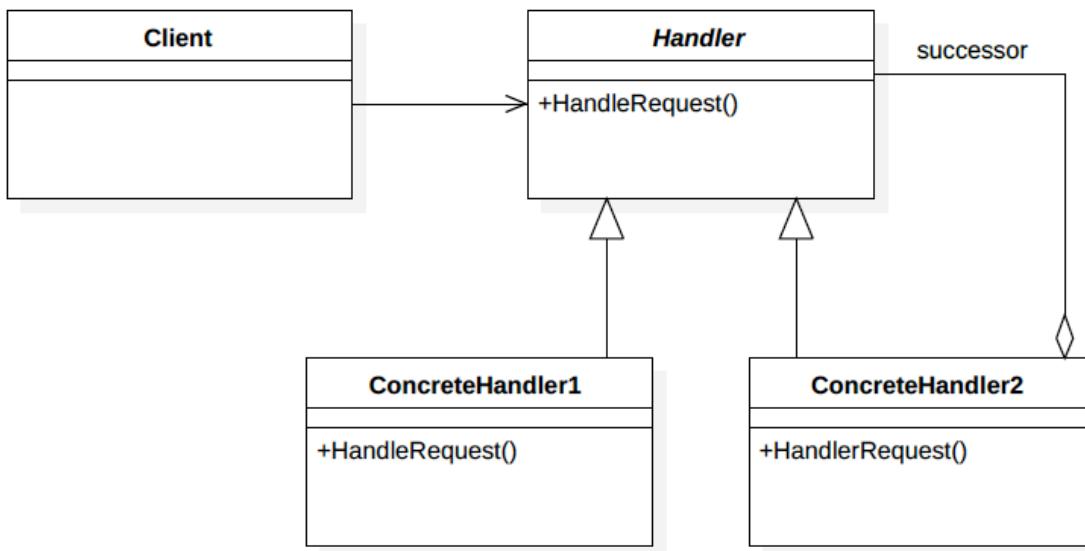
```

Decorator Example 01
Message : Let's enjoy!!!
Message : Let's enjoy!!!More!!!
Message : Let's enjoy!!!More!!!More!!!

```

[책임 연쇄 패턴(Chain of Responsibility)를 이용한 공유된 자원 이벤트의 처리]

책임 연쇄 패턴은 공유하고 있는 자원에서 발생하는 이벤트를 기다리는 다양한 핸들러를 만들 때 사용할 수 있다. 예를 들어, 어떤 이벤트 채널을 공유하는 상황에서 이벤트의 소스에 대해 여러가지 일을 해주어야 하지만, 해당 채널을 공유하는 경우 자신에게 맞는 이벤트인지 확인해서 각각의 경우에 맞게 처리해주어야 한다. 만약 자신이 기대하는 이벤트가 아니라면 다른 핸들러에게 이벤트를 전달해 주어야 한다. 이와 같은 처리 과정이 마치 체인(Chain)처럼 핸들러 들이 연결되어 있다고해서 이름 붙여졌다.



클라이언트는 발생한 이벤트의 처리를 핸들러에 맡기게 된다. 핸들러는 여러 개가 있을 수 있으며, 각각의 핸들러는 포인터를 이용해서 연결(Chain)된 형태를 가지고 있다. 즉, 핸들러("Handler")를 만족시키는 이벤트 핸들러의 연결된 리스트로 생각할 수 있을 것이다. 각각의 핸들러는 자신이 처리할 이벤트를 정의하고 있으며, 자신에게 해당된 이벤트가 아닌 경우에는 연결된 다른 핸들러로 처리를 위임하게 된다. 이때 각각의 핸들러는 클라이언트에 동일한 인터페이스를 제공하기 위해서 동일한 이름의 함수를 정의하고 있다("HandleRequest()").

```

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct message {
    unsigned int ID;
    char msg[ 100 ];
} MESSAGE;

typedef struct chain CHAIN;

struct chain {
    unsigned int ID;
    char *name;
    CHAIN *next;
    bool (*do_somthing)( CHAIN *target, MESSAGE *msg );
};

Client
  +Client()
  -~Client()

  void handleRequest()
      if (this->next != NULL)
          this->next->handleRequest();
      else
          do_something(this, msg);
  }

  void setNext( CHAIN *next )
      this->next = next;
  }

  CHAIN *getNext()
      return this->next;
  }

  void do_something( CHAIN *target, MESSAGE *msg )
      printf( "Chain %s handle request %d\n", target->name, target->ID );
  }

  void print()
      printf( "Chain %s handle request %d\n", name, ID );
  }
}
  
```

체인을 만들기 위해서 자료구조를 정의했다. 각각의 체인에 연결될 핸들러 들은 자신에게 해당하는 이벤트(메시지)가 전달되었을 때, 이를 처리하기 위한 함수도 같이 필드로 가지고 있다("do_something()"). 자신에게 전달된 메시지 인지를 확인하기 위해서 메시지와 핸들러의 ID를 비교하는데 사용한다. 만약, 자신에게 해당하는 메시지 아닌 경우에는 다음에 연결된(Chained) 핸들러를 호출해주기 위해서 "next" 필드를 가진다.

```
bool myHandler( CHAIN *target, MESSAGE *msg ) {
    if ( target->ID == msg->ID ) {
        printf( "Message Handler Name : %s, Message : %s\n", target->name,
                msg->msg );
        return true;
    }
    return false;
}
```

```
CHAIN handler1 = { 1, "1st Handler", NULL, myHandler };
CHAIN handler2 = { 2, "2nd Handler", &handler1, myHandler };
CHAIN handler3 = { 3, "3rd Handler", &handler2, myHandler };
```

체인의 연결고리 각각에 대해서 다른 핸들러 들을 사용해도 좋지만, 여기서는 구현을 단순화 시키기 위해서 하나의 핸들러 함수를 공통으로 사용하도록 했다. 체인의 순서도 중요한데, 자주 발생하는 이벤트의 핸들러는 체인의 앞쪽에 위치시키는 것이 좋다. 여기서는 단순히 "3->2->1"번의 순서로 체인을 만들어 주었다. 필요하다면 체인에 동적으로 핸들러 들을 추가하는 함수를 만들어도 될 것이다. 예를 들어, 리눅스(Linux)의 인터럽트 핸들러가 인터럽트를 공유하는 경우에는, 이와 같은 방식의 체인을 생성해서 사용하고 있으며, 핸들러의 추가와 삭제가 가능하도록 만든 인터페이스를 가지고 있다.

```
bool sendMessage( CHAIN *target, MESSAGE *msg ) {
    while( target ) {
        if( target->do_somthing( target, msg ) )
        {
            break;
        }
        target = target->next;
    }

    if ( target == NULL ) {
        printf( "Cannot handle the message : %d, %s\n", msg->ID, msg->msg );
        return false;
    }
    return true;
}
```

체인에 메시지를 던지기 위해서 구현한 함수다. 만약, 메시지를 처리한 핸들러가 있다면, 핸들러의 복귀값이 "true"가 될 것이다. 따라서, 나머지 핸들러 들에 메시지를 전달할 필요가 없기에 즉시 복귀한다. 자신이 처리해야 할 이벤트가 아닌 경우에는 "false"를 복귀값으로 돌려준다. 처리할 핸들러를 찾기 위해서, 체인을 하나씩 추적하며 각각의 연결된 핸들러 들을 호출한다.

```
int main(void) {
    MESSAGE message1 = { 1, "This Message is for the First!!!" };
    MESSAGE message2 = { 2, "This Message is for the Second!!!" };
    MESSAGE message3 = { 3, "This Message is for the Third!!!" };
```

```

puts("Chain of Responsibility Example 01");

sendMessage( &handler3, &message1 );
sendMessage( &handler3, &message2 );
sendMessage( &handler3, &message3 );

return EXIT_SUCCESS;
}

```

위의 코드는 각각의 핸들러 들이 자신에 해당하는 메시지를 제대로 처리하는지를 보기 위해서 메시지들을 정의해 주었다. 그것을 "sendMessage()" 함수를 이용해서 메시지 핸들러 들에게 전달했다. 아래는 프로그램을 실행한 결과이다.

[결과]

Chain of Responsibility Example 01

Message Handler Name : 1st Handler, Message : This Message is for the First!!!

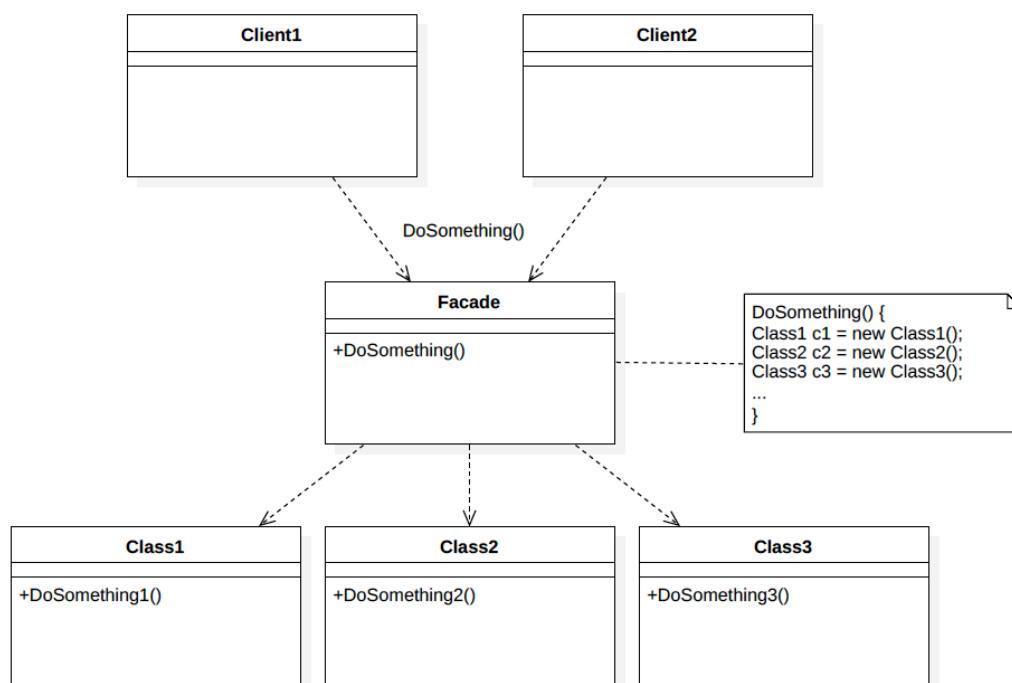
Message Handler Name : 2nd Handler, Message : This Message is for the Second!!!

Message Handler Name : 3rd Handler, Message : This Message is for the Third!!!

[파사드 패턴(Facade Pattern)을 이용한 내부 구현 감추기]

파사드는 건물의 출입구를 나타내는 말이다. 즉, 건물에 들어가기 위해서는 항상 사용해야 하는 문이다. 소프트웨어 구현에서도 특정 소프트웨어 패키지를 사용하기 위해서 반드시 통과해야 하는 인터페이스가 있다면, 이를 파사드라고 할 수 있겠다. 물론, 건물에 직접적으로 들어가는 것이 막혀 있듯이, 특정 패키지를 직접적으로 접근하는 것은 금지된다. 반드시 파사드에서 제공되는 API 만을 사용해서 접근해야 한다.

이와 같이 하는 것은 인터페이스를 기준으로 변경의 범위를 한정하는 것과 동시에, 변화에 대해서도 유연한 코드를 만들기 위해서다. 즉, 패키지 내부의 코드 변화가 인터페이스를 이용하는 클라이언트 코드에 영향을 주지 않도록 만드는 것이다. 파사드는 일종의 계약(Contract)관계로 계속 유지될 수 있다.



클라이언트는 실제 자신의 요청을 처리할 패키지(Package)의 알고 있지만, 구체적으로 요청이 어떻게 처리 될지는 알지 못한다. 단순히 처리가 파사드("Facade")가 제공하는 "DoSomething()"을 호출하는 것으로 가능하다는 것만 알고 있다. 따라서, 클라이언트들은 항상 파사드에게 요청을 전달하고, 실제적인 처리는 파사드가 패키지 내부의 구성요소들을 활용해서 처리해준다.

```
/* Facade.h */
#ifndef FACADE_H_
#define FACADE_H_

typedef struct computer_information {
    char *CPU;
    unsigned int GHz;
    unsigned int RAM;
    unsigned int HDD;
} COMPUTER_INFORMATION;

typedef struct computer COMPUTER;
struct computer {
    char *CPU;
    unsigned int GHz;
    unsigned int RAM;
    unsigned int HDD;

    void (*turnOn)(COMPUTER *comp);
    void (*turnOff)(COMPUTER *comp);
    COMPUTER_INFORMATION *(*getInfo)(COMPUTER *comp);
};

#endif __cplusplus
extern "C" {
#endif

extern COMPUTER *getComputer(void);
extern void turnOnComputer(COMPUTER *comp);
extern void turnOffComputer(COMPUTER *comp);
extern COMPUTER_INFORMATION *getComputerInformation(COMPUTER *comp);
extern void print_computer_information(COMPUTER_INFORMATION *info);

#endif /* FACADE_H_ */
```

앞의 코드는 파사드에서 제공하는 자료구조와 API에 대한 인터페이스를 보여준다. 이곳에 정의된 자료구조의 구체적인 부분들도 숨길 수 있지만, 여기서는 구현을 단순화 시키기 위해서 공개하도록 만들었다. 사용하는 측에서는 타입만 보고 내부의 구현에 의존적인 코드를 작성 해서는 안될 것이다. 나중에 "main()" 함수의 구현에서 볼 수 있듯이, 제공되는 자료의 타입과 API 만을 사용해서 원하는 작업을 완료 할 수 있다.

```
/* Facade.c */
```

```
#include <stdio.h>
#include "Facade.h"

extern COMPUTER myComputer;

COMPUTER *getComputer(void) {
    return &myComputer;
}

void turnOnComputer(COMPUTER *comp) {
    comp->turnOn(comp);
}

void turnOffComputer(COMPUTER *comp) {
    comp->turnOff(comp);
}

COMPUTER_INFORMATION *getComputerInformation(COMPUTER *comp) {
    return comp->getInfo(comp);
}

void print_computer_information(COMPUTER_INFORMATION *info) {
    printf("The CPU : %s\n", info->CPU);
    printf("Clock : %dGHz\n", info->GHz);
    printf("HDD : %dGByte\n", info->HDD);
    printf("RAM : %dGByte\n", info->RAM);
}
```

파사드 자체가 일종의 계층 역할을 한다. 즉, 건물 내부로 진입하는 진입점(Entry Point)과 같은 역할만 한다. 따라서, 파사드에서 제공하는 인터페이스는 책임을 내부의 구현으로 위임(Delegation)하는 역할만 수행하도록 했다. 진짜 구현은 내부에 정의된 실제 인터페이스에서 수행하도록 한다.

```
/* FacadeImplementation.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Facade.h"

static void myTurnOn(COMPUTER *comp) {
    printf("Computer %s is Turned On!!!\n", comp->CPU);
}

static void myTurnOff(COMPUTER *comp) {
    printf("Computer %s is Turned Off!!!\n", comp->CPU);
}

static COMPUTER_INFORMATION *myGetInfo(COMPUTER *comp) {
    COMPUTER_INFORMATION *info = NULL;

    if ((info = (COMPUTER_INFORMATION *) malloc(sizeof(COMPUTER_INFORMATION))) == NULL) {
```

```

        printf("Cannot allocate memory for computer information!!!\n");
    }

    if ((info->CPU = (char *) malloc(strlen(comp->CPU) + 1)) == NULL) {
        printf("Cannot allocate memory for CPU information!!!\n");
    }

    if (info == NULL) {
        return NULL;
    }

    if (info->CPU == NULL) {
        free(info);
        return NULL;
    }

    strncpy(info->CPU, comp->CPU, strlen(comp->CPU) + 1);
    info->GHz = comp->GHz;
    info->RAM = comp->RAM;
    info->HDD = comp->HDD;
    return info;
}

```

```
COMPUTER myComputer = { "Intel Centrino QuadCore", 3, 8, 500, myTurnOn,
                        myTurnOff, myGetInfo };
```

앞의 코드는 인터페이스와 구현을 분리된 계층으로 만들어서 위임하는 구조로 구현했다. 구체적인 기능들은 실제 구현이 맡게된다. 따라서, 이곳에서 일어나는 변경은 파사드를 사용하는 측에서는 가려지게 되며, 변화의 효과도 계약을 위반하지 않는 수준에서 영향을 줄 것이다. 즉, 계약의 변경(계약의 변경이란 인터페이스가 하는 역할이 달라진다는 것을 의미) 까지는 가지않게 된다. 따라서, 내부의 자료구조나 코드의 변경이 외부의 코드 변경으로 이어지지 않는다.

```

/* main.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Facade.h"

int main(void) {
    COMPUTER *computer = getComputer();
    COMPUTER_INFORMATION *info = getComputerInformation( computer );

    turnOnComputer( computer );
    turnOffComputer( computer );
    print_computer_information( info );

    return EXIT_SUCCESS;
}

```

사용하는 측에서는 파사드에서 제공되는 API만을 사용할 수 있다. 직접적으로 내부의 API를 사용하지는 못한다. 내부 API들은 전부 "static"으로 한정되어 사용되기에, 외부 파일에서는 접근하는 것이 불가

능하다. 내부의 서비스를 이용하기 위해서는 반드시 패스드를 통해서 접근해야만 한다. 프로그램을 실행한 결과는 아래와 같다.

[결과]

Facade Example 01

Computer Intel Centrino QuadCore is Turned On!!!

Computer Intel Centrino QuadCore is Turned Off!!!

The CPU : Intel Centrino QuadCore

Clock : 3GHz

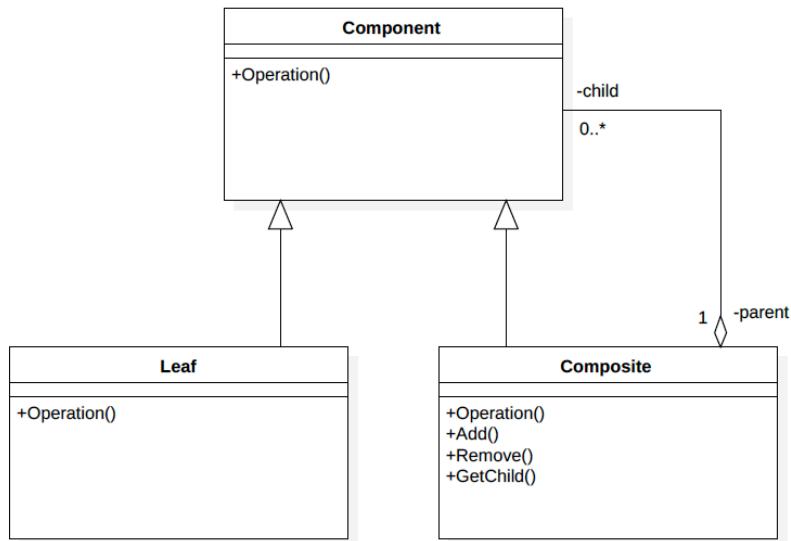
HDD : 500GByte

RAM : 8GByte

[컴포짓 패턴(Composite Pattern)을 이용한 자료구조의 묶음 다루기]

"Composite"이란 여러 개의 자료구조를 하나로 묶은 것이다. 단순히 자료구조만 있는 것이 아니라, 그에 관련된 함수들 까지 같이 묶어서, 각각의 자료구조에 해당하는 핸들러들을 가지고 있다. 이렇게 묶인 자료구조들이 마치 하나인 것 처럼 다룰 수 있도록 만들어주는 것으로, 얼마나 많은 자료구조를 내부에 가지고 있을지는 알지 못하지만 외부로 보여지는 인터페이스를 통해서 이를 하나로 취급할 수 있게 된다.

하나로 묶이는 자료구조는 동일한 인터페이스를 제공한다고 가정할 수 있다. 물론, 내부적으로는 그 동일한 인터페이스의 구현 내용은 달라질 수 있지만, 외부에서 보면 그것은 마치 하나의 자료형(Data Type)으로 생각해 볼 수 있다. 따라서, 내부의 구현이 변경되거나 새로운 구성요소가 추가 되더라도, 외부에서는 일관되게 처리할 수 있다. 따라서, 내부 구성요서의 변경에 대해서 외부의 코드 변경은 발생하지 않는다.



컴포짓 패턴은 하나의 일을 처리하기 위해서 다양한 구성요소가 협동을 하는 관계로 이루어 진다. 클라이언트는 컴포넌트의 대표("Component")에게 일을 의뢰하고, 처리는 내부를 구성하고 있는 더 작은 컴포넌트들의 합집합이 담당하게 되는 것이다. 내부의 컴포넌트는 추가와 삭제가 가능하며, 자신이 관리하는 컴포넌트들을 관리할 수 있는 인터페이스를 가진다. 또한, 각각의 컴포넌트는 하위에 자신이 관리하는 컴포넌트들을 가질 수 있으며, 부모와 자식간에 양방향 포인터를 유지한다. 각 컴포넌트는 자식이 없을 수는 있지만, 최상위 컴포넌트를 제외하면 모두 부모 컴포넌트에 대한 연결을 가진다.

```
#ifndef COMPOSITE_H_
#define COMPOSITE_H_
```

```

typedef struct composite COMPOSITE;
typedef struct message MESSAGE;
typedef struct element ELEMENT;

struct message {
    unsigned int ID;
    char *message;
};

struct element {
    unsigned int ID;
    void (*do_something)( MESSAGE *msg );
};

#ifndef __cplusplus
extern "C" {
#endif

extern COMPOSITE *get_composite( void );
extern void add_element( COMPOSITE *comp, ELEMENT *element );
extern void delete_element( COMPOSITE *comp, ELEMENT *element );
extern void do_something( COMPOSITE *comp, MESSAGE *msg );

#ifndef __cplusplus
}
#endif
#endif /* COMPOSITE_H_ */

```

전달되는 메시지는 ID를 가지고 있으며, 구성요소(Element)들도 자신의 ID를 가진다. 구성요소들의 ID는 나중에 해당 구성요소 들을 관리하기 위해서 사용된다. 각각의 구성요소 들은 자신이 어떻게 메시지를 처리할지를 결정하는 핸들러("do_something()")을 가진다. 따라서, 구성요소를 추가할 경우에는 메시지에 대한 핸들러도 같이 추가해 주어야 할 것이다.

전체 뮤음을 얻기 위한 함수("get_composite()")와 추가 및 삭제하기 위한 함수들을 정의했다. 그리고, 자료구조의 뮤음을 순환하면서 각각의 설치된 구성요소 들에게 메시지를 전달해줄 인터페이스도 정의했다("do_something()").

```

#include <stdio.h>
#include "Composite.h"
#define MAX_ELEMENT 100

struct composite {
    ELEMENT *comp[ MAX_ELEMENT ];
    void (*add)(COMPOSITE *comp, ELEMENT *element);
    void (*delete)( COMPOSITE *comp, ELEMENT *element );
    void (*do_something)(COMPOSITE *comp, MESSAGE *msg);
};

static void myAdd(COMPOSITE *comp, ELEMENT *element) {
    if ((element->ID < 0) || (element->ID >= MAX_ELEMENT)) {
        printf("The element ID %d is out of range!!!\n", element->ID);
    }
}

```

```

        return;
    }

    if (comp->comp[element->ID] != NULL) {
        printf("The element ID %d is already used!!!\n", element->ID);
        return;
    }

    comp->comp[element->ID] = element;
    return;
}

static void myDelete(COMPOSITE *comp, ELEMENT *element) {
    if ((element->ID < 0) || (element->ID >= MAX_ELEMENT)) {
        printf("The element ID %d is out of range!!!\n", element->ID);
        return;
    }

    if (comp->comp[element->ID] == NULL) {
        printf("The element ID %d is already unused!!!\n", element->ID);
        return;
    }

    comp->comp[element->ID] = NULL;
    return;
}
static void myDo_something(COMPOSITE *comp, MESSAGE *msg) {
    ELEMENT *element;

    for (int i = 0; i < MAX_ELEMENT; i++) {
        element = comp->comp[i];
        if (element != NULL) {
            if (element->do_something != NULL) {
                element->do_something(msg);
            }
        }
    }
}

```

각각의 구성요소들은 자신이 가지는 ID값을 이용해서 배열에 저장된다. 이미 저장되어 있다면 “NULL” 이외의 값을 가질 것이다. 저장을 위해서는 ID에 해당하는 배열에 구성요소(Element)의 주소를 넣어주기만 하면 된다. 삭제는 반대로 ID에 해당하는 배열에 “NULL” 값을 넣어주는 것으로 처리했다. 각각의 구성요소들에 메시지를 전달하기 위해서 단순히 반복문을 돌면서, “NULL”이 아닌 구성요소에 메시지를 전달했다("do_something()").

```
COMPOSITE myComposite = {{ NULL }, myAdd, myDelete, myDo_something };
```

```
COMPOSITE *get_composite( void ) {
    return &myComposite;
}
```

```

void add_element( COMPOSITE *comp, ELEMENT *element ) {
    comp->add( comp, element );
}

void delete_element( COMPOSITE *comp, ELEMENT *element ) {
    comp->delete( comp, element );
}

void do_something( COMPOSITE *comp, MESSAGE *msg ) {
    comp->do_something( comp, msg );
}

```

구현상 편의를 위해서 미리 선언된 뮤음(Composite)을 사용했다. 추가 및 삭제, 메시지 처리에 대한 함수의 포인터를 이용해서 자료구조를 초기화 했다. 만약 필요하다면, 뮤음을 얻는 ("get_composite()") 함수에 메모리 공간을 할당해서 초기화 하는 코드를 넣어도 될 것이다. 구성요소를 추가하고 삭제하기 위해서 외부에 공개 되는 함수는 별도로 작성했지만, 책임을 위임(Delegation)하는 방법으로 구현했다.

```

#include <stdio.h>
#include <stdlib.h>

#include "Composite.h"

void print_message( MESSAGE *msg ) {
    printf("The Message : %s\n", msg->message );
}

int main(void) {
    puts("Composite Example 01"); /* prints Composite Example 01 */
    MESSAGE msgHelloWorld = { 1, "Hello, World!!!" };
    MESSAGE msgByeWorld   = { 2, "Bye, World!!!" };

    ELEMENT elm01 = { 1, print_message };
    ELEMENT elm02 = { 2, print_message };
    ELEMENT elm03 = { 3, print_message };

    COMPOSITE *myComposite = get_composite();
    add_element( myComposite, &elm01 );
    add_element( myComposite, &elm02 );
    add_element( myComposite, &elm03 );

    do_something( myComposite, &msgHelloWorld );
    do_something( myComposite, &msgByeWorld );

    delete_element( myComposite, &elm01 );
    delete_element( myComposite, &elm02 );
    delete_element( myComposite, &elm03 );

    return EXIT_SUCCESS;
}

```

테스트 프로그램에서는 먼저 묶음(Composite)을 얻어낸 후, 각각의 구성요소(Element)를 삽입했다. 각각의 구성요소들은 공통된 핸들러를 가지며("print_message()"), 나중에 전달받은 메시지를 처리하기 위해서 호출 된다. 테스트 프로그램에서는 각각의 구성요소를 직접 선언해서 사용했지만, 실제로 코드를 만들 경우에는 별도의 파일로 분리한 다음 초기화하는 방식으로 처리하는 것이 좋을 것이다. 즉, 사용하는 측(클라이언트)에서는 구성요소의 구조에 대해서 모르는 편이 더 좋다.

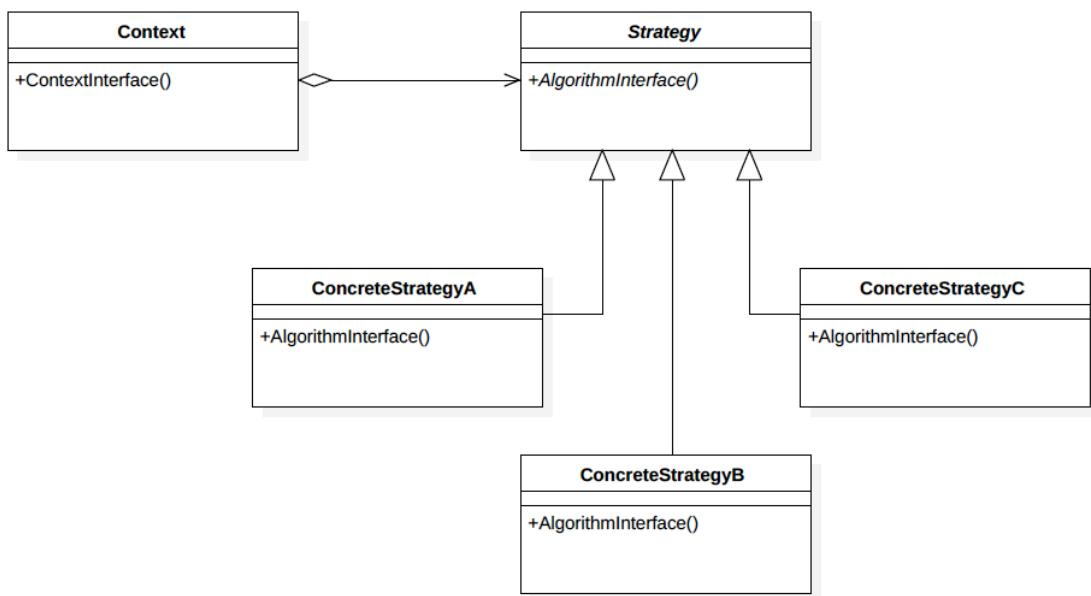
메시지의 경우에는 전달하는 측에서도 같이 알아야 하겠지만, 나머지 자료구조들은 숨기는 것이 유리하다. 나중에 구성요소나 묶음의 내부 구현이 변경 되더라도, 일관된 인터페이스에 의해서 사용할 수 있다면 변경의 범위를 줄일 수 있다.

[결과]

```
Composite Example 01
The Message : Hello, World!!!
The Message : Hello, World!!!
The Message : Hello, World!!!
The Message : Bye, World!!!
The Message : Bye, World!!!
The Message : Bye, World!!!
```

[전략 패턴(Strategy Pattern)을 활용한 알고리즘 변경]

전략(Strategy) 패턴은 동일한 인터페이스를 공유하는 알고리즘 들이 클라이언트 측의 요청 형태에 맞게 처리하도록 만들 때 사용할 수 있다. 즉, 클라이언트의 요청 타입에 맞는 서비스를 제공해 줄 목적으로 사용 되어, 클라이언트 코드의 변경없이 새로운 방법으로 알고리즘을 변경할 수 있도록 허락한다. 이것도 마찬가지로 변경을 최소화 하는 기법으로 인터페이스에 대한 재활용이라고 볼 수 있다.



전략 패턴은 선택적으로 알고리즘과 같은 “전략(Strategy)”을 적용할 수 있도록 하기 위해서, 대표적인 자료구조(“Strategy”)를 정의하며, 해당 자료구조를 제공하는 어떤 알고리즘도 추가될 수 있는 자료구조를 내부에 유지하고 있다. 따라서, 특정 상황에 적합한 전략을 선택해서 이미 알고 있는 인터페이스 (“AlgorithmInterface()”)를 이용해서 요청을 전달하면 된다.

여기서는 메시지의 타입에 따라 각각 개별적으로 처리할 수 있는 핸들러를 제공하는 방법을 예로 들었다. 즉, 새로운 메시지의 타입이 정의 되더라도, 그것을 이용하는 클라이언트의 입장에서는 동일한 인터

페이스를 통해서 요청을 보내게 되며, 새로운 타입에 대한 핸들러를 추가하는 것도 인터페이스만 유지한다면 간단히 추가할 수 있다.

```
#ifndef STRATEGY_H_
#define STRATEGY_H_

typedef enum {
    MSG_TYPE_00 = 0,
    MSG_TYPE_01,
    MSG_TYPE_02,
    MSG_TYPE_03,
    MSG_TYPE_NONE
} MSG_TYPE;

typedef struct message {
    MSG_TYPE type;
    char *body;
} MESSAGE;

typedef struct strategy STRATEGY;

#ifdef __cplusplus
extern "C" {
#endif

extern STRATEGY *get_strategy( void );
extern void handle_message( STRATEGY *strategy, MESSAGE *msg );

#ifdef __cplusplus
}
#endif
#endif /* STRATEGY_H_ */
```

전략 패턴을 구현하기 위해서 구조체("struct")를 정의 했으며, 클라이언트의 요청을 제공받는 인터페이스에 전달하기 위해서 "MESSAGE" 타입을 정의해서 사용하고 있다. 사용하는 전략 패턴을 클라이언트 측에서 내부구조를 모르고 사용할 수 있도록 일종의 핸들 역할을 하는 것을 먼저 얻어오도록(생성하도록) 인터페이스를 제공한다("get_strategy()"). 메시지 처리를 위해서는 "handle_message()"함수를 생성한 핸들을 이용해서 호출하면 된다.

```
#include <stdio.h>
#include "Strategy.h"
typedef void (*HANDLER)(MESSAGE *msg);

struct strategy {
    HANDLER handle[MSG_TYPE_NONE];
};

static void print_message_handler(MESSAGE *msg) {
    printf("The Message Type : %d, body : %s\n", msg->type, msg->body);
}
```

```

STRATEGY myStrategy = { .handle = { print_message_handler,
    print_message_handler, print_message_handler, print_message_handler } };

STRATEGY *get_strategy(void) {
    return &myStrategy;
}

void handle_message(STRATEGY *strategy, MESSAGE *msg) {
    if ((msg->type < MSG_TYPE_00) || (msg->type >= MSG_TYPE_NONE)) {
        printf("Cannot handle message!!!\n");
    }
    strategy->handle[msg->type](msg);
}

```

여기서는 구현을 간단하게 하기 위해서, 각각의 메시지 타입에 대해서 동일한 핸들러를 만들어 주었다 ("myStrategy"). 각각의 타입에 대한 핸들러를 호출하는 부분은 간단히 배열을 이용해서 전달받은 메시지를 다시 전달하는 방식으로 처리했다. 각각의 메시지는 자신의 타입에 대한 것과 메시지 내용을 화면에 출력하는 것으로 처리를 마치도록 했다.

```

#include <stdio.h>
#include <stdlib.h>
#include "Strategy.h"

int main(void) {
    puts("Strategy Pattern Example 01"); /* prints Strategy Pattern Example 01 */
    MESSAGE msg00 = { MSG_TYPE_00, "This is 0th message!!!" };
    MESSAGE msg01 = { MSG_TYPE_01, "This is 1st message!!!" };
    MESSAGE msg02 = { MSG_TYPE_02, "This is 2nd message!!!" };
    MESSAGE msg03 = { MSG_TYPE_03, "This is 3rd message!!!" };

    STRATEGY *strategy = get_strategy();
    handle_message( strategy, &msg00 );
    handle_message( strategy, &msg01 );
    handle_message( strategy, &msg02 );
    handle_message( strategy, &msg03 );

    return EXIT_SUCCESS;
}

```

각각의 메시지 타입이 어떻게 처리가 되는지를 보기 위해서 다양한 메시지들을 정의했다. 초기에 핸들러들을 가지고 있는 핸들을 얻기 위해서, "get_strategy()"를 호출 했으며, 각각의 메시지를 핸들러에 전달했다. 결과에서 보듯이 각각의 메시지는 자신의 메시지 타입에 해당하는 핸들러 들에 의해서 처리가 되는 것을 볼 수 있을 것이다. 만약, 새로운 메시지 타입이 추가 되더라도, 간단히 핸들러를 추가하는 방법으로 기능을 확장할 수 있게된다. 핸들러들을 정의하는 것은 응용에 맞게 변경하면 되지만, 핸들러의 인터페이스는 동일하게 항상 유지되어야 할 것이다. 즉, 핸들러가 제공하는 계약에 맞추면 쉽게 새로운 기능(알고리즘)을 추가하거나 새로운 알고리즘으로 대체할 수 있다.

[결과]

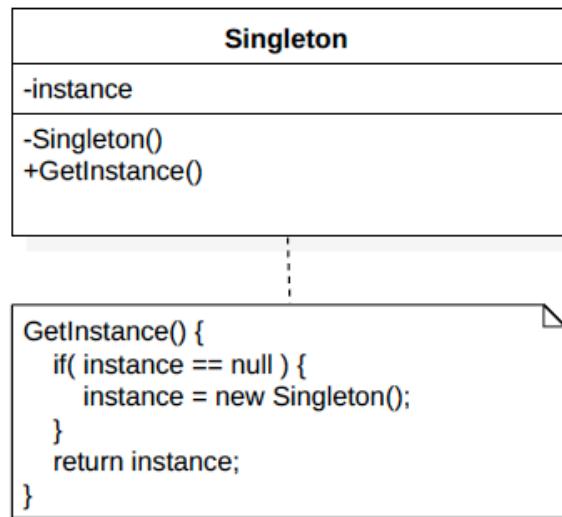
Strategy Pattern Example 01
The Message Type : 0, body : This is 0th message!!!
The Message Type : 1, body : This is 1st message!!!

The Message Type : 2, body : This is 2nd message!!!

The Message Type : 3, body : This is 3rd message!!!

[단일자 패턴(Single Pattern)을 이용한 공유 자원 다루기]

단일자(싱글턴: Singleton) 패턴은 프로그램에서 단지 하나만 존재하는 자료구조(자원)를 만들고 싶을 때 사용하는 방법이다(물론, 여러개의 한정된 자원을 가지는 것으로 확장이 가능하다). 즉, 두 개 이상이 동일한 자료구조를 생성하지 못하도록 만드는 방법이다. 이와 같은 것이 필요한 상황은 시스템의 한정된 자원을 표현할 때이다. 예를 들어 특정한 자원이 하나가 있을 경우, 이를 공유하기 위한 방법으로 사용할 수 있다.



시스템에 단 하나의 인스턴스(Instance ; 실제 메모리에 존재하는 오브젝트)를 만들기 위해서 자신의 생성을 단 한 번만 허락한다. 그 외의 생성 요청에 대해서는 이미 만들어진 인스턴스를 돌려주면 된다. 따라서, 시스템에는 항상 원하는 개수의 인스턴스(여기서는 한 개만)를 유지할 수 있다. 외부에서 직접적인 인스턴스 접근을 막기 위해서, 자신이 관리하는 인스턴스는 감추고 있는 형태(Private : “-“)로 만들었다.

한 가지 아쉬운 것은 시스템에 하나만 존재한다는 것을 전역 변수의 또 다른 표현으로 혼동해서 사용할 수 있다는 점이다. 특히, 객체지향 프로그래밍을 하는 경우, 흔히 전역 변수 대용으로 싱글턴을 사용하는 경우가 있다. 하지만, 전역 변수를 사용하면 공유하는 코드들은 독립성을 유지하기 어렵기에 될 수 있으면 사용을 자제하는 것이 좋다.

이런 식으로 하나 둘 씩 전역 변수들이 늘어나다 보면, 전역 변수에 대한 의존적인 코드가 늘어나게 되어 전체 코드가 마치 한 덩어리처럼 분리할 수 없는 상태에 도달하게 된다. 이는 구조적으로 시스템의 확장성과 변경성, 유지보수성을 떨어뜨려 고비용(수정 및 추가에 드는 비용) 저효율(일정 지연, 추가 개발자 투입 등)의 시스템을 만들 가능성성이 높다. 물론, 직접적인 접근에 의해서 일부 얻을 수 있는 성능상의 장점은 있을지도 모르지만, 전역 변수는 대부분 최적화에 악영향을 준다는 점을 기억해야 한다.

```
#ifndef SINGLETON_H_
#define SINGLETON_H_
```

```
typedef struct singleton SINGLETON;
```

```
#ifdef __cplusplus
extern "C" {
```

```
#endif

extern SINGLETON *create_singleton( void );
extern SINGLETON *get_Instance( SINGLETON *singleton );
extern void destroy_singleton( SINGLETON **singleton );

#endif __cplusplus
}
#endif
#endif /* SINGLETON_H_ */
```

싱글턴을 이용하기 위해서는 싱글턴을 생성하는 것이 있어야 한다. 그리고, 생성은 단지 한 번만 수행되어야 한다. 즉, 여러 번의 생성 요청이 있다라도 단지 하나만 만들어야 하고, 나머지 요청에 대해서는 이미 만들어진 것을 사용하도록 해주면 된다. 이를 위해서 "create_singleton()" 함수가 정의 되었다.

생성된 자료구조를 사용하기 위해서 "get_instance()" 함수를 정의 했지만, 사실상 앞에서 생성한 자료구조의 포인터를 공유하기에, 생성을 제외하고는 동일한 동작을 있다고 볼 수 있다. "destroy_singleton()"은 생성된 싱글턴을 삭제하기 위해서 사용한다.

여기서 보여주는 코드는 약간 어색하기는 하지만, 어쨌든 필요한 자료구조를 단지 한번만 생성하도록 만들어 시스템 전체적으로는 하나의 자료구조의 인스턴스(일반적으로 실제 메모리를 차지하는 것, 혹은 특정한 자원을 표시하기 위해서 필요한 메모리)만 남게된다. 물론, 각각의 자료구조에 대해서 상속과 같은 개념을 C언어에서는 사용하지 못하기에 별도로 자료구조를 선언해 주어야 한다. 각각의 자료구조는 자신만을 위한 생성기(Create)를 갖추고 있어야 하며, 나중에 사용이 끝났을 때 이를 삭제(Delete)해주는 것도 가지고 있어야 한다.

```
#include <stdio.h>
#include <stdlib.h>
#include "Singleton.h"

struct singleton {
    SINGLETON *instance;
    SINGLETON *(*getInstance)(SINGLETON *singleton);
    void (*destroy)(SINGLETON **singleton);
};

SINGLETON *mySingleton = NULL;

static SINGLETON *getInstance(SINGLETON *singleton) {
    return singleton->instance;
}

static void destroySingleton(SINGLETON **singleton) {
    if (*singleton != mySingleton) {
        printf("Cannot destroy resources!!!\n");
        return;
    }

    free(mySingleton);
    *singleton = NULL;
    return;
}
```

```

}

SINGLETON *create_singleton(void) {
    if (mySingleton != NULL) {
        return mySingleton;
    }

    if ((mySingleton = (SINGLETON *) malloc(sizeof(SINGLETON))) == NULL) {
        printf("Cannot create resources!!!\n");
        return NULL;
    }

    mySingleton->instance = mySingleton;
    mySingleton->getInstance = getInstance;
    mySingleton->destroy = destroySingleton;
    return mySingleton;
}

```

내부적으로 싱글턴을 관리하기 위해서 사용 되는 함수들을 만들어 주었다. 즉, 싱글턴 자료구조의 형태는 동일하지만, 할당과 제거를 달리 처리할 수 있도록 만들어 주었다. 따라서, 다른 타입이라고 하더라도 이곳에 일반화된 자료구조를 정의하고, 각각의 형태에 맞는 생성 함수와 제거 함수를 정의해서 사용하면 될 것이다. 동일한 인터페이스를 이용해서 자료구조 마다 달리 정의된 생성 함수와 제거 함수를 접근할 수 있게 된 것이다.

```

SINGLETON *get_Instance( SINGLETON *singleton ) {
    return singleton->getInstance( singleton );
}

void destroy_singleton( SINGLETON **singleton ) {
    if ( *singleton == NULL )
    {
        printf( "Cannot destroy already destroyed singleton!!!\n" );
        return;
    }
    (*singleton)->destroy( singleton );
    return;
}

```

앞의 코드는 싱글턴을 사용하는 측에 필요한 인터페이스를 제공하기 위한 부분이다. 즉, 싱글턴 내부의 정의된 자료형에 의존하지 않도록 만들어주고 있다. 각각의 클라이언트 요청은 실제 구현된 싱글턴에서 제공하는 함수 들에 의해서 처리될 것이다.

```

#include <stdio.h>
#include <stdlib.h>
#include "Singleton.h"

int main(void) {
    puts("Singleton Example 01"); /* prints Singleton Example 01 */
    SINGLETON *singleton = NULL;

    singleton = create_singleton(); /* 여러번 생성 요청을 하더라도 하나 밖에 얻지 못한다. */

```

```

singleton = create_singleton();
singleton = get_Instance( singleton ); /* 공유 자원을 얻는다. */
singleton = get_Instance( singleton );
destroy_singleton( &singleton );
destroy_singleton( &singleton ); /* 이미 삭제된 싱글턴은 다시 삭제할 수 없다. */

return EXIT_SUCCESS;
}

```

위의 코드는 싱글턴의 생성과 인스턴스 얻기, 삭제 등을 여러 번 실행해서 정말 하나의 싱글턴 만이 생성되고 사용되는지를 확인하기 위한 것이다. 마지막에 싱글턴을 두 번 삭제하려고 하면, 하나만 있기 때문에 두 번째 요청은 실패하는 것을 보여줄 것이다.

[결과]

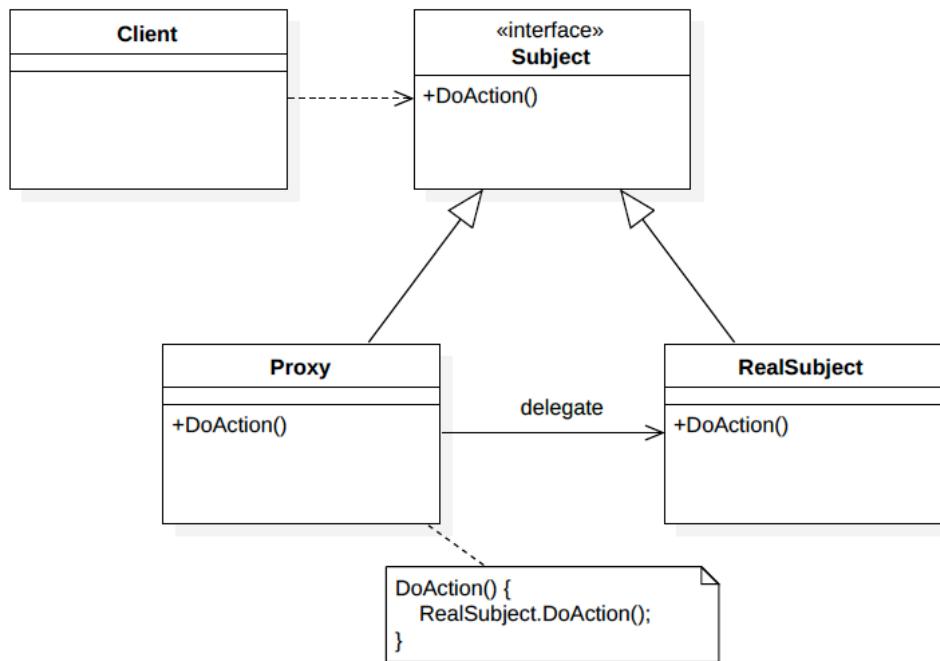
Singleton Example 01

Cannot destroy already destroyed singleton!!!

물론, 이와 같이 전역에서 사용될 자료구조를 감싸서(Wrapping) 제공하는 것이 성능에 악 영향을 줄 수 있다고 이야기 할 수 있다. 하지만, 인터페이스를 이용해서 상세 구현에 의존하는 부분을 분리시켜 다양한 자료구조를 일관된 방법으로 하나만 존재할 수 있도록 만드는 것은 의미 있는 일이다. 하지만, 전역 변수를 전부 이렇게 처리하는 것은 바람직하지 않으며, 전역 변수를 될 수 있으면 줄이려는 노력을 해야 한다. 자료구조의 사용범위를 한정하고 해당 자료구조에 접근하는 인터페이스를 만들어서 제공하는 것은 변경의 영향을 국지화(Localization)시키는데 있어서 중요한 부분이다.

[대리자 패턴(Proxy Pattern)을 이용한 권한의 위임]

대리자(Proxy) 패턴은 요청을 대신 처리해주는 역할을 맡기기 위해서 사용한다. 즉, 직접적인 처리를 담당하는 쪽은 누구인지는 모르지만, 일을 맡겨야 할 경우 대리자를 만들어서 요청을 전달하는 것이다. 요청은 대리자를 통해서 처리를 담당하는 쪽에 전달되고, 처리된 결과도 대리자를 통해서 전달받을 수 있다. 즉, 대리자를 사이에 두고 요청을 보내는 측과 요청을 처리하는 측이 분리될 수 있으며, 각각의 코드의 변경이 상대방에 대해서 영향을 주지 않도록 분리된다. 변경이 필요하다면 대리자를 수정하면 될 것이다.



클라이언트는 자신이 사용할 주제(“Subject”)에 대해서 알고 있지만, 그것을 실제로 누가 처리 할지는 알지 못한다. 즉, 대리자(“Proxy”)를 통해서 처리를 의뢰할 뿐이다. 실제 처리는 대리자가 알고 있는 “RealSubject”에 처리를 위임(Delegate)하는 것으로 이루어진다. 대리자와 “RealSubject”는 동일한 인터페이스를 제공해 주어야 하기에, “RealSubject”는 동일한 인터페이스를 제공한다면 다른 것으로 교체도 가능하다.

```

#ifndef PROXY_H_
#define PROXY_H_

typedef struct message MESSAGE;

struct message {
    char *msg;
};

typedef struct subject SUBJECT;
typedef struct proxy PROXY;

#ifdef __cplusplus
extern "C" {
#endif

extern PROXY *getProxy( void );
extern void sendRequest( PROXY *proxy, MESSAGE *msg );

#ifdef __cplusplus
}
#endif
#endif /* PROXY_H_ */
    
```

프록시를 사용하기 위해서는 일단 프록시를 먼저 얻어야 한다("getProxy()"). 최소한의 정보만 주기 위해서, 전달할 수 있는 메시지 포맷을 정의 했으며, 이를 통해서 프록시에 요청을 보낼 수 있다 ("sendRequest()"). 프록시를 사용하는 측에서는 메시지의 형식만 알뿐이지, 프록시가 어떻게 메시지를 처리해주는지에 대해서는 관심이 없다. 즉, 필요한 정보만 공개하고 나머지 상세한 구현은 인터페이스 넘어로 숨긴 것이다.

```
#include <stdio.h>
#include "Proxy.h"
struct subject {
    char *name;
    void (*handler)(SUBJECT *subject, MESSAGE *msg);
};

struct proxy {
    char *name;
    SUBJECT *subject;
    void (*handler)(PROXY *proxy, MESSAGE *msg);
};

static void subject_handler(SUBJECT *subject, MESSAGE *msg) {
    printf("Subject Handler : %s, %s\n", subject->name, msg->msg);
}

static void proxy_handler(PROXY *proxy, MESSAGE *msg) {
    printf("Proxy Handler : %s, %s\n", proxy->name, msg->msg);
    proxy->subject->handler(proxy->subject, msg);
}

SUBJECT mySubject = { "My Subject", subject_handler };
PROXY myProxy = { "My Proxy", &mySubject, proxy_handler };

PROXY *getProxy(void) {
    return &myProxy;
}

void sendRequest(PROXY *proxy, MESSAGE *msg) {
    proxy->handler(proxy, msg);
}
```

프록시는 내부에서 요청을 누가 처리해 줄지 알고 있어야 한다. 여러 개의 처리를 담당하는 자료구조 ("SUBJECT")를 정의할 수도 있지만, 간단히 하나만 정의해서 사용하도록 했다. 프록시는 전달받은 요청을 단순히 자신이 관리하는 핸들러 들에게 넘겨주는 역할만 담당한다. 여기서도 요청을 처리하는 측은 요청의 전달 형태만 관심이 있을 뿐, 프록시가 호출 했는지, 누가 호출 했는지는 알지 못한다. 따라서, 프록시를 기준으로 요청하는 측과 요청을 처리하는 측은 분리되어 관리될 수 있다는 것을 알 수 있다. 따라서, 한 쪽의 코드 변화가 다른 한 쪽의 코드 변화를 일으키지 않게 만들어 준다.

```
#include <stdio.h>
#include <stdlib.h>
#include "Proxy.h"

int main(void) {
```

```

puts("Proxy Example 01");

PROXY *proxy = getProxy();
MESSAGE mssage = { "Hello, World!!!" };

sendRequest( proxy, &mssage );

return EXIT_SUCCESS;
}

```

위의 코드는 프록시를 얻어와서 자신이 전달하고 싶은 메시지를 정의한 다음, 프록시에게 요청을 의뢰하고 있다. 즉, 요청하는 측은 프록시가 모든 처리를 마치고 결과를 돌려줄 것이라고만 알고 있을 뿐이다. 실제로 그 요청을 누가 처리해줄지는 관심이 없다. 프로그램의 실행 결과는 아래와 같다

[결과]

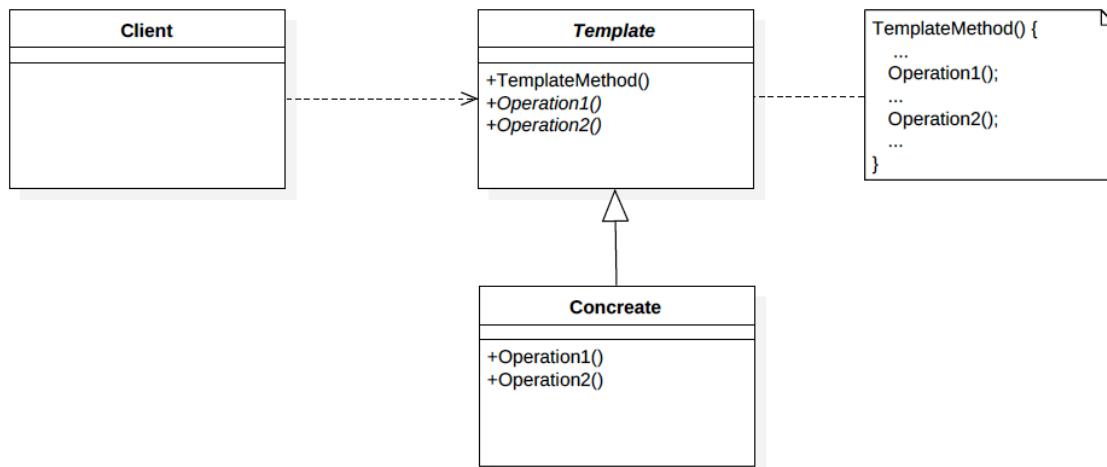
Proxy Example 01

Proxy Handler : My Proxy, Hello, World!!!

Subject Handler : My Subject, Hello, World!!!

[템플릿 메소드 패턴(Template Method Pattern)을 활용한 다양성의 구현]

템플릿 메서드는 말 그대로 틀을 제공하는 방법이다. 즉, 세세한 구현은 달라질 수 있지만, 흐름은 템플릿처럼 만들어 전체적인 모양은 동일하게 유지하도록 만들어준다. 세세한 방법들의 인터페이스는 동일하며, 그 인터페이스를 사용하는 흐름의 순서를 유지하는 것이다. 예를 들어, 정렬 알고리즘을 구현할 때, 각각의 정렬 대상에 대한 비교 방법은 달라지지만 배치하는 방법은 동일한 경우다. 사람의 정보를 가지고 정렬할 때, 사람의 키나 나이, 몸무게, 사는 지역, 연령등 다양한 방법을 통해서 순서를 정할 수 있다. 이때, 그 비교를 하기 위한 각각의 정보에 대해서 다른 방법을 제공해야 하지만, 동일한 인터페이스로 구체적인 것을 가릴 수 있게된다.



일의 일반화된 처리 절차를 정의하는 것을 템플릿("Template")으로 보고, 그것의 구체적인 구현을 하위 모듈(Module:여기서는 "Concrete")을 이용하는 구조는 나중에 템플릿을 따르는 다른 모듈로 쉽게 대체가 가능하다. 즉, 인터페이스만 동일하면 실제 구현("Concrete")은 달라져도 사용하는 측의 코드는 변경되지 않는다. 템플릿 내부에서 실제 구현을 이용해서 처리해 주기 때문이다.

```
#include <stdio.h>
#include <stdlib.h>
```

```
typedef struct algorithm ALGORITHM;

struct algorithm {
    int (*sum)( const int operand1, const int operand2 );
    int (*subtract)( const int operand1, const int operand2 );
    int (*multiply)( const int operand1, const int operand2 );
    int (*divide)( const int operand1, const int operand2 );
};
```

앞의 코드는 템플릿에서 사용하게 되는 인터페이스를 제공하는 자료구조다. 즉, 이 자료구조를 만족시키면, 템플릿을 사용하는 측에서는 구체적인 구현의 차이를 알지 못한다. 따라서, 동일한 인터페이스만 제공하면 다른 것으로 교체해도 된다는 뜻이다(물론, 기대하는 역할이 달라지면 안되지만). 새로운 알고리즘을 구현하고자 한다면, 이 인터페이스에 해당하는 함수들을 새로 정의해서 제공해 주면 될 것이다.

```
static int sum( const int operand1, const int operand2 ) {
    return operand1 + operand2;
}

static int subtract( const int operand1, const int operand2 ) {
    return operand1 - operand2;
}

static int multiply( const int operand1, const int operand2 ) {
    return operand1 * operand2;
}

static int divide( const int operand1, const int operand2 ) {
    if( operand2 == 0 )
    {
        printf( "Cannot calculate the result!!!\n" );
        exit( 1 );
    }
    return operand1 / operand2;
}
```

```
ALGORITHM myAlgorithm = { sum, subtract, multiply, divide };
```

여기서 제공하는 코드는 간단한 예제이다. 템플릿이 사용하는 인터페이스를 제공하기 위해서 간단한 사칙연산을 수행하는 함수 들로 구성했다. 템플릿은 여기서 제공되는 함수들을 가지고, 자신의 함수를 만들것이다. 따라서, 여기에 있는 함수의 내부 코드를 변경 하더라도, 사용하는 측의 코드 변경은 없으나 결과는 제공되는 연산의 결과에 따라 달라질 수 있다. 내부의 구체적인 구현의 변화에 대해서 다양한 결과를 낼 수 있다는 점에서, 사용자 측의 코드를 변경하지 않고도 새로운 기능을 추가할 수 있게된다.

```
typedef struct template TEMPLATE;

struct template {
    ALGORITHM *algo;
    int ( *method )( const ALGORITHM *algo, const int operand1, const int operand2 );
};

static int calculate( const ALGORITHM *algo, const int operand1, const int operand2 ) {
```

```

        return algo->sum( operand1, operand2 ) + algo->subtract( operand1, operand2 )
        + algo->multiply( operand1, operand2 ) + algo->divide( operand1, operand2 );
}

```

```
TEMPLATE myTemplate = { &myAlgorithm, calculate };
```

템플릿은 자신의 인터페이스를 구현하기 위해서 필요한 인터페이스들을 알고 있어야한다("algo"). 그리고, 템플릿이 원하는 동작을 수행하기 위해서 미리 알고 있는 인터페이스들을 이용했다("calculate()"). 여기서는, 구현의 단순화를 위해서 템플릿 변수를 선언했으나 실제 구현에서는 메모리에서 할당한 후에 초기화하는 방법을 사용할 수도 있다.

```

int main(void) {
    puts("Template Method Pattern Example 01");
    printf( "The calculation result : %d\n", myTemplate.method( myTemplate.algo, 1,
2 ) );
    printf( "The calculation result : %d\n", myTemplate.method( myTemplate.algo, 10,
20 ) );
    printf( "The calculation result : %d\n", myTemplate.method( myTemplate.algo, 20,
10 ) );
    printf( "The calculation result : %d\n", myTemplate.method( myTemplate.algo, 0,
0 ) ); /* Cannot calculate the result!!! */
    return EXIT_SUCCESS;
}

```

앞의 코드는 템플릿에서 제공하는 방법을 이용해서 계산 결과를 출력하는 것을 보여준다. 템플릿은 제공 받은 인터페이스를 이용해서 일을 처리하지만, 그 구체적인 내용에 대해서는 관여하지 않는다. 인터페이스들이 사용되는 과정(흐름)만을 담당할 뿐이다. 새로운 구현이 필요하다면, 템플릿에 전달될 구체적인 함수들의 인터페이스만 다른다면, 상위 코드의 흐름에는 영향을 주지않고 기능 확장이나 변경이 가능하다.

[결과]

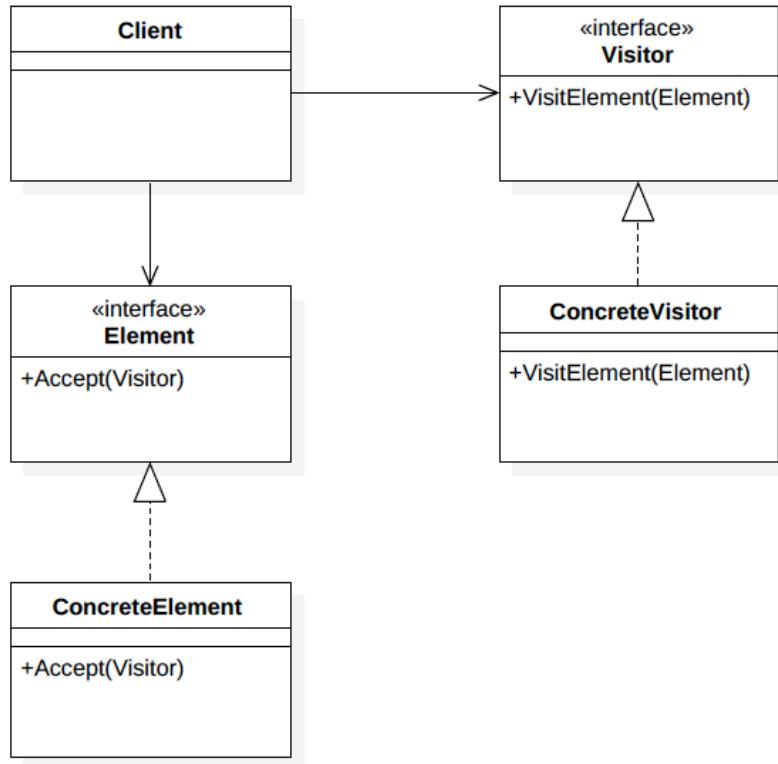
```

Template Method Pattern Example 01
The calculation result : 4
The calculation result : 220
The calculation result : 242
Cannot calculate the result!!!

```

[방문자 패턴(Visitor Pattern)을 이용한 연결된 자료구조의 원소 다루기]

비지터 패턴은 연결된 형태의 자료구조의 각 원소(Element)들을 방문(Visit)해서 특정한 일을 처리할 때 유용한 방법이다. 이때, 사용자의 입장에서는 각 원소의 연결된 구조에 대한 내부적인 상세 정보는 알 필요가 없으며, 연결된 자료구조와 이를 방문하는 비지터 간에 해야할 역할을 구분해서 정하기만 하면 된다.



클라이언트는 특정 인터페이스를 따르는 구성 요소("Element")들을 알고 있으며, 그 구성 요소들을 일일이 방문해서 처리를 맡고 있는 방문자("Visitor")도 알고 있다. 각각의 구성 요소들은 클라이언트가 방문자를 넘겨줄 때, 넘겨 받은 방문자를 이용하는 방법을 정의하고 있다("Accept()").

비지터는 자신의 타입과 자신이 어떤 일을 할지를 각 원소들의 정의된 자료형에 반영해 주어야 하며, 각 원소들의 자료구조는 비지터를 받아들여(Accept), 자신이 해주고 싶은 일을 정의해 주면 된다. 비지터는 각 원소들을 방문해서, 각 원소들이 하고 싶어하는 일을 처리(호출)하거나, 혹은 자신이 원하는 일을 수행할 수 있다.

```

#include <stdio.h>
#include <stdlib.h>

typedef struct visitor VISITOR;
typedef struct element ELEMENT;

struct element {
    char *name;
    ELEMENT *left;
    ELEMENT *right;
    void (*accept)( ELEMENT *element, VISITOR *visitor );
    void (*do_something)( ELEMENT *element );
};

struct visitor {
    void (*visit)( ELEMENT *element );
};

  
```

각 원소들이 가져야 하는 자료구조는 연결을 유지할 수 있는 링크(Link)와 비지터를 받아들이는 함수 인터페이스("accept()"), 비지터가 방문했을 때 할 일을 등이다("do_something()"). 비지터는 각각의 원

소를 방문해서 원소들이 원하는 일을 수행하게 된다.

```
void accept( ELEMENT *element, VISITOR *visitor ) {
    /* Visit Left Child —> Visit Right Child —> Visit Current */
    if ( element->left != NULL )
    {
        (element->left)->accept( element->left, visitor );
    }

    if ( element->right != NULL )
    {
        (element->right)->accept( element->right, visitor );
    }
    visitor->visit( element );
}
```

각 원소들이 비지터를 받아들이는 것은 "accept()" 함수에서 처리한다. 여기서는 먼저 자신에게 링크된 다른 원소들이 있는지를 확인하고, 만약 있다면 각각의 원소 들에 대해서도 동일한 "accept()"를 호출한다. 하위 원소 들에 대한 처리가 완료 되었을 때, 자신을 처리하기 위해서 비지터의 "visit()" 함수를 호출해 주었다.

여기서 만약 필요하다면, 자신의 원소를 먼저 방문하도록 변경할 수도 있다. 혹은, 위의 예에서 우측(Right)원소를 먼저 방문하고, 좌측을 방문하도록 변경할 수도 있다. 각각의 원소마다, 자신의 하위 원소 들에 대한 방문 방법을 스스로 정할 수 있기에, 일반적으로 "Tree"와 같은 형태의 자료구조를 알고리즘에서 처리하는 것과는 다르게 처리할 수 있다는 것을 기억하기 바란다.

```
void print_element( ELEMENT *element ) {
    printf( "The Element Name : %s\n", element->name );
}

void visit( ELEMENT *element ) {
    element->do_something( element );
}
```

각 원소를 방문해서 할 일은 각 원소들이 가지고 있는 "do_something()" 함수를 실행하는 일이다. 여기서는 단순히 각각의 원소들이 가지고 있는 이름을 화면상에 출력하도록 만들어 주었다. 필요하다면, 이곳에 각각의 원소가 하고 싶은 일을 정의해서, 비지터가 이를 실행할 수 있도록 만들어주면 될 것이다.

```
ELEMENT element_left_left = {"Element Left Left", NULL, NULL, accept, print_element };
ELEMENT element_left_right = {"Element Left Right", NULL, NULL, accept,
                             print_element };
ELEMENT element_right_left = {"Element Right Left", NULL, NULL, accept,
                             print_element };
ELEMENT element_right_right = {"Element Right Right", NULL, NULL, accept,
                             print_element };
ELEMENT element_left = { "Element Left", &element_left_left, &element_left_right, accept,
                        print_element };
ELEMENT element_right= { "Element Right", &element_right_left, &element_right_right,
                        accept, print_element };
ELEMENT element_root = { "Element root", &element_left, &element_right, accept,
```

```
    print_element );
```

```
VISITOR myVisitor = { visit },
```

코드 구현을 간단하게 만들기 위해서, 각 원소들의 연결구조를 정적으로 정의해 주었다. 실제 응용에서는 이와 같은 자료구조를 동적으로 생성할 수 있는 코드들을 추가할 필요가 있다. 비지터도 마찬가지로 정적으로 이곳에서 초기화 해 주었다.

```
int main(void) {
    puts("Visitor Pattern Example 01"); /* prints Visitor Pattern Example 01 */

    element_root.accept( &element_root, &myVisitor );
    return EXIT_SUCCESS;
}
```

예제에서 가장 상위의 루트(Root) 원소가 비지터를 받아들이는("accept()") 함수를 호출하는 것으로 모든 원소에 대한 방문을 시작하도록 만들었다. 아래는 예제를 실행한 결과를 보여준다. 결과에서 볼 수 있듯이, 현재는 각 원소를 왼쪽에서부터 깊이 우선 방향으로 하나씩 방문자가 돌아다니고(Traverse) 있는 것을 확인할 수 있다.

[결과]

```
Visitor Pattern Example 01
The Element Name : Element Left Left
The Element Name : Element Left Right
The Element Name : Element Left
The Element Name : Element Right Left
The Element Name : Element Right Right
The Element Name : Element Right
The Element Name : Element root
```

앞에서 본 코드를 조금 수정해서 새로운 기능을 추가해 보도록 하자. 이번에는 각각의 원소가 가지고 있는 값의 총 합을 구하는 것을 만들겠다. 앞에서 만들어진 코드에 추가해서, 새로운 방문자를 정의해서 이 방문자가 각각의 원소가 가지는 값(Value)의 총 합을 구해서 보여주는 것으로 변경했다.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct visitor VISITOR;
typedef struct element ELEMENT;

struct element {
    char *name;
    ELEMENT *left;
    ELEMENT *right;
    unsigned int value; /* Added for recording element's value */
    void (*accept)(ELEMENT *element, VISITOR *visitor);
    void (*do_something)(ELEMENT *element);
};

struct visitor {
    unsigned int sum; /* Added for summation of element's value */
    void (*visit)(VISITOR *visitor, ELEMENT *element);
```

```

};

void accept(ELEMENT *element, VISITOR *visitor) {
    if (element->left != NULL) {
        (element->left)->accept(element->left, visitor);
    }
    if (element->right != NULL) {
        (element->right)->accept(element->right, visitor);
    }
    visitor->visit(visitor, element);
}

void print_element(ELEMENT *element) {
    printf("The Element Name : %s\n", element->name);
}

void AVisit(VISITOR *visitor, ELEMENT *element) {
    element->do_something(element);
}

void BVisit(VISITOR *visitor, ELEMENT *element) {
    visitor->sum += element->value; /* Added for visitor to make a summation of
element's value */
}

/* We added new field. So, we have to define each elements' values here */
ELEMENT element_left_left = { "Element Left Left", NULL, NULL, 100, accept,
                             print_element };
ELEMENT element_left_right = { "Element Left Right", NULL, NULL, 10, accept,
                             print_element };
ELEMENT element_right_left = { "Element Right Left", NULL, NULL, 1000, accept,
                             print_element };
ELEMENT element_right_right = { "Element Right Right", NULL, NULL, 200, accept,
                             print_element };
ELEMENT element_left = { "Element Left", &element_left_left,
                        &element_left_right, 20, accept, print_element };
ELEMENT element_right = { "Element Right", &element_right_left,
                        &element_right_right, 2000, accept, print_element };
ELEMENT element_root = { "Element root", &element_left, &element_right, 10000,
                        accept, print_element };

VISITOR myVisitor1 = { 0, AVisit };
VISITOR myVisitor2 = { 0, BVisit }; /* Second visitor to make a summation of element's
value */

int main(void) {
    puts("Visitor Pattern Example 01");

    element_root.accept(&element_root, &myVisitor1);
    element_root.accept(&element_root, &myVisitor2); /* New visitor */
    printf("The Total Value : %d\n", myVisitor2.sum);
}

```

```

    return EXIT_SUCCESS;
}

```

코드에서 보듯이 간단히 원소의 필드를 추가하고, 새로운 방문자의 "visit()" 함수를 재정의 하는 것으로 자료구조의 내부에 포함된 각각의 원소가 가지는 값을 전부 더했다. 새로운 방문자가 방문을 완료한 후, 각각의 원소들이 가지는 전체 값이 얼마인지를 보여주는 코드도 추가했다. 프로그램의 실행 결과는 아래와 같다.

결과에서 알 수 있듯이, 새로운 방문자를 정의해서 각각의 원소 들에 대해서 새로운 동작을 할 수 있다는 것이 이 패턴의 장점이다. 즉, 연결된 자료구조에 새로운 기능을 추가하는 것이 쉬우며, 기존의 클라이언트("main()")에 최소한의 변경으로 원하는 것을 할 수 있다는 점이다. 물론, 클라이언트가 사용할 인터페이스는 항상 동일하게 유지되고 있다.

[결과]

Visitor Pattern Example 01

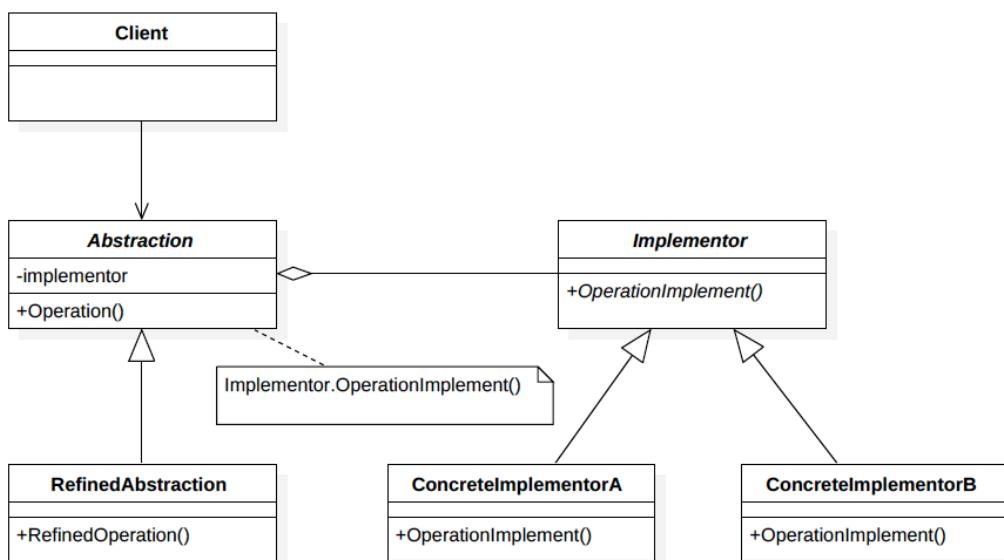
```

The Element Name : Element Left Left
The Element Name : Element Left Right
The Element Name : Element Left
The Element Name : Element Right Left
The Element Name : Element Right Right
The Element Name : Element Right
The Element Name : Element root
The Total Value : 13330

```

[브릿지 패턴(Bridge Pattern)을 이용한 모듈간의 연결고리 만들기]

브리지 패턴은 말 그대로 차이가 있는 두 모듈 간에 다리(Bridge)를 만드는 것이다. 실제 구현을 숨기고, 가교 역할을 하는 자료구조와 인터페이스를 정의해서 활용하는 방법이다. 실제의 구현은 사용자 측의 코드를 연결하기 위한 인터페이스와 맞추어주어야 하며, 이를 다른 모듈의 구현으로 대체하는 방식으로 의존성을 분리한다. 따라서, 다리를 기준으로 양측의 모듈은 독립적으로 내부구조를 변경할 수 있다.



클라이언트는 자신이 사용할 인터페이스("Abstraction")을 알고 있으며, 인터페이스 자체는 다시 실체적인 구현이 달라질 수 있다("RefinedAbstraction"). 실제 구현된 인터페이스는 자신이 처리해야 할 일을 구현자('Implementator')를 통해서 처리할 수 있다는 것을 알며, 실체화된 구현자는 여러 개가 있을

수 있다("ConcreteImplementatorA, ConcreteImplementatorB"). 따라서, 클라이언트와 클라이언트가 이용하는 인터페이스의 구현자, 실제 구현자 등이 각각 정해진 인터페이스만 준수한다면, 서로 의존하지 않고 변경될 수 있다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct message MESSAGE;

struct message {
    char *msg;
};

typedef struct bridgeImpl BRIDGEIMPL;

struct bridgeImpl {
    void (*do_operation)( MESSAGE *msg );
};

typedef struct bridge BRIDGE;

struct bridge {
    BRIDGEIMPL *impl;
    void (*do_operation)( BRIDGE *bridge, MESSAGE *msg );
};
```

브릿지 자체와 구현을 분리해서, 브릿지 내부에 브릿지의 구체적인 구현 부분을 숨겨두었다 ("BRIDGE"). 브릿지의 실제 구현은 "BRIDGEIMPL"에 의해서 이루어지며, 이 타입만 만족할 수 있다면 다른 모듈의 구현으로 대체가 가능하다.

메시지 타입은 단순히 데이터의 전달을 위해서 만든 것으로 파라미터의 전달을 목적으로 한다. 따라서, 실제 구현에서는 원하는 형태로 변경해서 사용하면 될 것이다.

```
static void do_operation(BRIDGE *bridge, MESSAGE *msg) {
    bridge->impl->do_operation(msg);
}

static void print_message(MESSAGE *msg) {
    printf("The Message : %s\n", msg->msg);
}

static void add_and_print_message(MESSAGE *msg) {
    char temp[100] = "Hello, World!!!";
    strncat(temp, msg->msg, strlen(msg->msg) + 1);
    printf("The Message : %s\n", temp);
}
```

```
BRIDGEIMPL bridgeImpl01 = { print_message };
BRIDGEIMPL bridgeImpl02 = { add_and_print_message };
BRIDGE bridge01 = { &bridgeImpl01, do_operation };
```

```
BRIDGE bridge02 = { &bridgeImpl02, do_operation };
```

구현을 단순화 하기 위해서 브릿지의 실제 구현은 단순히 메시지를 화면 상에 표시하는 것으로 만들었다. 각각의 브릿지마다 다른 브릿지의 구현을 연결해서 사용하고 있지만, 실제로는 둘을 서로 바꾸어 지정해도 동일한 방법으로 사용할 수 있다는 것을 알 수다. 따라서, 브릿지 타입만 사용자 측에 보여주고 실제 구현은 변경이 쉬워진다.

```
int main(void) {
    puts("Bridge Pattern Example 01"); /* prints Bridge Pattern Example 01 */
    MESSAGE msg = { "Bye, Cruel World!!!" };

    bridge01.do_operation( &bridge01, &msg );
    bridge02.do_operation( &bridge02, &msg );
    return EXIT_SUCCESS;
}
```

위의 코드는 브릿지를 이용해서 실제로 어떻게 동작하는지를 보여주는 간단한 예이다. 만약, 브릿지의 완전한 자료구조까지 감추고 싶다면, 헤더 파일과 구현 파일을 브릿지와 브릿지의 실제구현으로 나눌 수 있을 것이다. 그리고, 사용자 측에는 브릿지의 타입 정보와 사용하기 위한 API정도만 개방하면 된다. 프로그램의 실행 결과는 아래와 같다.

[결과]

```
Bridge Pattern Example 01
The Message : Bye, Cruel World!!!
The Message : Hello, World!!!Bye, Cruel World!!!
```

이번에는 위에서 실제로 브릿지를 이용할 때, 하나의 브릿지만 이용해서 브릿지의 실제 구현을 동적으로 변경하는 경우를 보도록 하겠다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct message MESSAGE;

struct message {
    char *msg;
};

typedef struct bridgeImpl BRIDGEIMPL;

struct bridgeImpl {
    void (*do_operation)(MESSAGE *msg);
};

typedef struct bridge BRIDGE;

struct bridge {
    BRIDGEIMPL *impl;
    void (*do_operation)(BRIDGE *bridge, MESSAGE *msg);
};
```

```

static void do_operation(BRIDGE *bridge, MESSAGE *msg) {
    bridge->impl->do_operation(msg);
}

static void print_message(MESSAGE *msg) {
    printf("The Message : %s\n", msg->msg);
}

static void add_and_print_message(MESSAGE *msg) {
    char temp[100] = "Hello, World!!!";
    strncat(temp, msg->msg, strlen(msg->msg) + 1);
    printf("The Message : %s\n", temp);
}

/* Changed Bridge Implementations */
BRIDGEIMPL bridgeImpl01 = { print_message };
BRIDGEIMPL bridgeImpl02 = { add_and_print_message };

/* Added to change bridge implementation dynamically */
void set_bridge_implementation(BRIDGE *bridge, BRIDGEIMPL *bridgeImpl) {
    bridge->impl = bridgeImpl;
}

```

변경된 부분은 브릿지의 정적인 자료구조를 없애고, 브릿지의 구현만 두 개 남겨두었다. 나중에 동적으로 브릿지의 구현을 변경하기 위해서 만들었으며, 실제 구현에서는 동적으로 메모리를 할당하는 방법으로 변경할 수도 있을 것이다. 대신에 브릿지의 구현을 브릿지에 추가하는 "set_bridge_implementation()" 함수가 들어갔다.

```

int main(void) {
    puts("Bridge Pattern Example 01"); /* prints Bridge Pattern Example 01 */
    MESSAGE msg = { "Bye, Cruel World!!!" };
    BRIDGE bridge = { NULL, do_operation };

    set_bridge_implementation( &bridge, &bridgeImpl01 );
    bridge.do_operation( &bridge, &msg );
    set_bridge_implementation( &bridge, &bridgeImpl02 );
    bridge.do_operation( &bridge, &msg );

    return EXIT_SUCCESS;
}

```

위의 코드에서 보듯이, 브릿지의 실제 구현을 동적으로 변경할 수 있게 되었다. 마찬가지로 브릿지 자체의 자료구조를 보여주고 싶지 않다면, 새로운 API를 추가해서 "bridge.do_operation()"과 같은 브릿지 타입의 내부에 정의된 함수 포인터를 가리면 될 것이다.

[결과]

```

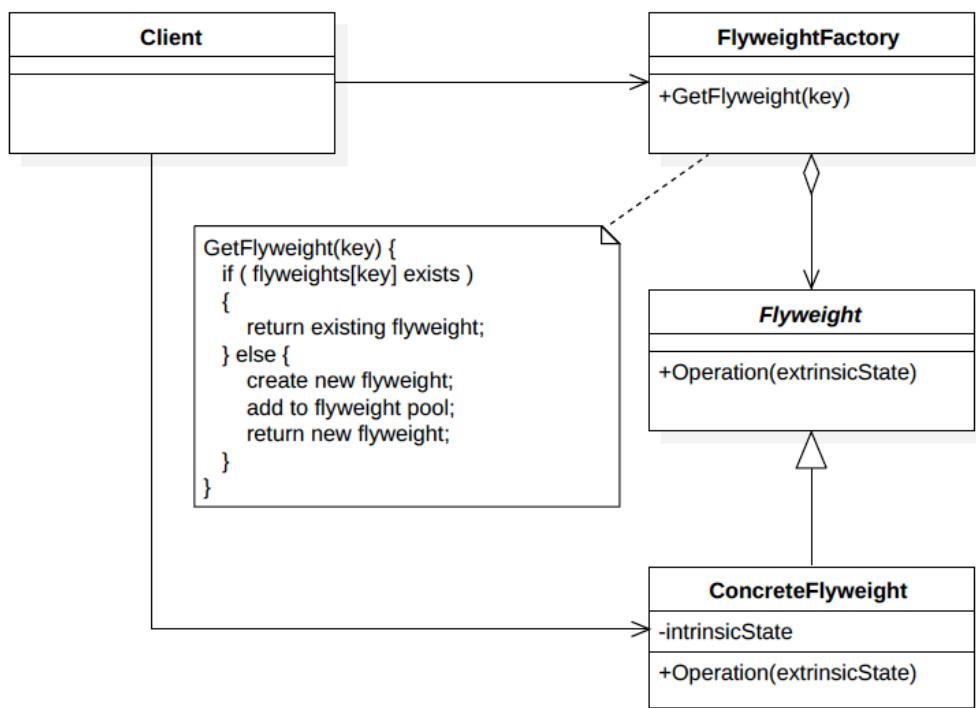
Bridge Pattern Example 01
The Message : Bye, Cruel World!!!
The Message : Hello, World!!!Bye, Cruel World!!!

```

[플라이 웨이트 패턴(Fly Weight Pattern)을 이용한 작은 자료구조들의 생성]

플라이 웨이트 패턴(일명 경량 패턴)은 작은 자료구조를 대량으로 생산할 경우에 사용하는 방법이다. 즉, 전체 정보를 가지는 크기가 큰 자료구조를 생성하기 보다, 필요한 정보만 골라서 추린 작은 자료구조를 생성하고, 실제 그것에 대한 처리가 필요한 경우에만 한정해서 관련된 큰 자료구조를 접근하는 방식을 취하는 것이다.

예를 들어, 큰 해상도를 가지는 사진들을 전부 한번에 모아서 보여주기는 힘들다. 한 번에 큰 사진들을 전부 디코딩(Decoding)해서 보여주려면 시간도 많이 걸릴 것이다. 하지만, 작은 썸네일(Thumbnail)들을 만들어서 보여주다가 선택된 경우에만 실제 사진을 불러 와서 보여주는 것은 어렵지도 않고 처리 시간도 절약된다. 이와 같은 것을 구현하기 위해서 요약된 정보만을 가지는 작은 자료구조를 많이 생성하고, 필요한 경우에만 원본 자료구조에 접근하는 방식이 플라이 웨이트 패턴이다.



클라이언트는 작은 자료구조들의 생성과 관리를 맡고 있는 “FlyweightFactory”를 알고 있으며, “FlyweightFactory”에서는 “Flyweight”的 생성을 담당하고 있다. 이때 생성된 “Flyweight”들은 클라이언트의 요청을 처리할 수 있는 연산(“Operation”)을 정의하고 있으며, 내부 상태와 넘겨받는 외부 상태(“intrinsicState, extrinsicState”)에 따라 정의된 동작을 수행한다. 이미 생성되어 있는 “Flyweight”에 대해서는 기존에 존재하는 “Flyweight”를 생성 요청에 대해서 돌려주면 되고, 없다면 새로 만들어서 “Flyweight”를 관리하는 풀(Pool)에 넣어준 후, 생성된 “Flyweight”를 돌려주면 된다.

예제 코드는 구현을 간략화 하기 위해서 동일한 유형의 자료구조가 있을 경우에 이를 그대로 이용하고, 만약 실제로 어떤 일을 처리해야 할 경우에만 자료구조가 정의한 함수를 사용하도록 구현했다. 즉, 필요한 경우 이외에는 전체 자료구조를 이용할 필요가 없다.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

typedef struct messasge {

```

```

        char *msg;
} MESSAGE;

#define MAX_POOL_SIZE 10
#define MAX_NAME_LENGTH 100

typedef struct flyweight FLYWEIGHT;
typedef struct flyweight_factory FLYWEIGHT_FACTORY;

struct flyweight {
    char key[MAX_NAME_LENGTH]; /* Flyweight intrinsic state */
    bool empty;
    void (*operation)(char *key, MESSAGE *extrinsic_state);
};

struct flyweight_factory {
    FLYWEIGHT pool[MAX_POOL_SIZE];
    FLYWEIGHT *(*create)(FLYWEIGHT_FACTORY *factory, char *key);
    FLYWEIGHT *(*getFlyWeight)(FLYWEIGHT_FACTORY *factory, char *key);
};

static void print_flyweight_information(char *name, MESSAGE *extrinsic_state) {
    printf("The flyweight information : %s, %s\n", name, extrinsic_state->msg);
}

FLYWEIGHT noneFlyWeight = { "None FlyWeight", false, print_flyweight_information };

```

플라이 웨이트 패턴은 작은 자료구조들을 생성하고 관리하는 플라이 웨이트 팩토리와 플라이 웨이트 구조체로 이뤄진다. 플라이 웨이트 구조체는 자신의 내부(Intrinsic) 상태를 가지는 "key"를 필드를 가지고 있으며, 구조체의 풀(pool)에서 사용되지 않고 비었는지를 알려주는 "empty"필드, 플라이 웨이트 자체가 외부 상태(Extrinsic)에 반응하는 "operation()"함수 등을 가진다. 여기서는 단순히 플라이 웨이트의 정보와 전달받은 외부 상태를 출력해 준다.

플라이 웨이트의 팩토리는 플라이 웨이트의 풀(pool)과 플라이 웨이트의 생성을 담당하는 "create()", 특정 플라이 웨이트를 가져올 수 있는 "getFlyweight()"함수를 필드로 가진다.

```

static FLYWEIGHT *create(FLYWEIGHT_FACTORY *factory, char *key) {
    FLYWEIGHT *temp;
    int i;

    /* Find key */
    for (i = 0; i < MAX_POOL_SIZE; i++) {
        temp = &(factory->pool[i]);
        if (temp->empty == true) {
            continue;
        }
        if (strncmp(key, temp->key, strlen(key) + 1) == 0) {
            /* Found */
            printf("Found same flyweight!!!\n");
            return temp;
        }
    }
}

```

```

    }

/* Find empty slot */
for (i = 0; i < MAX_POOL_SIZE; i++) {
    temp = &(factory->pool[i]);
    if (temp->empty == true) {
        break;
    }
}

/* Check end of pool */
if (i >= MAX_POOL_SIZE) {
    printf("Cannot allocate flyweight in pool!!!\n");
    return &noneFlyWeight;
}

/* Initialize fields */
strncpy(temp->key, key, strlen(key) + 1);
temp->empty = false;
temp->operation = print_flyweight_information;

printf("Flyweight created: %d\n", i);
return temp;
}

```

플라이 웨이트의 생성은 이미 생성된 플라이 웨이트가 있는지 "key"를 가지고 검색한다. 만약, 이미 생성된 것이 있다면, 그것을 그대로 재활용한다. 만약, 없다면 새로운 플라이 웨이트를 생성하기 위해서 풀(Pool)에서 빈 곳을 찾는다. 모든 공간이 사용중이라면, "noneFlyWeight"의 포인터를 돌려준다. 빈 공간이 있을 경우에는 플라이 웨이트의 정보를 기록해주고, 접근할 수 있는 포인터를 돌려준다.

```

FLYWEIGHT *getFlyWeight(FLYWEIGHT_FACTORY *factory, char *key) {
    FLYWEIGHT *temp = NULL;

    /* Find key */
    for (int i = 0; i < MAX_POOL_SIZE; i++) {
        temp = &(factory->pool[i]);
        if (temp->empty == true) {
            continue;
        }
        if (strncmp(key, temp->key, strlen(key) + 1) == 0) {
            /* Found */
            printf("Found same flyweight!!!\n");
            return temp;
        }
    }
    printf("Cannot find flyweight!!!\n");
    return &noneFlyWeight;
}

```

```

void init_flyweight_factory(FLYWEIGHT_FACTORY *factory) {
    for (int i = 0; i < MAX_POOL_SIZE; i++) {

```

```

        factory->pool[i].empty = true;
    }
}

FLYWEIGHT_FACTORY myFlyweightFactory = { { }, create, getFlyWeight };

```

생성된 플라이웨이트를 찾는 것도 "key"값을 비교하는 것으로 이뤄진다. 만약 찾을 수 없다면, "nonFlyWeight"의 포인터를 전달한다. "NULL"값은 호출한 측에서 처리해주지 않아도 된다. 초기에 플라이 웨이트들의 풀(Pool)을 비었다고 표시하기(초기화) 위해서 "init_flyweight_factory()"를 호출해 주어야 하지만, 만약, 풀이 전부 "NULL"로 선언되고, 플라이 웨이트들을 동적으로 생성한다면, 이 함수는 필요하지 않다. 여기서는 구현 편의상 만들었을 뿐이다.

```

int main(void) {
    MESSAGE extrinsic_state = { "Hello, My FlyWeight!!!" };
    puts("Fly Weight Pattern Example 01");

    init_flyweight_factory(&myFlyweightFactory); /* Make Flyweight pool empty */
    myFlyweightFactory.create(&myFlyweightFactory, "myFlyWeight00");
    myFlyweightFactory.create(&myFlyweightFactory, "myFlyWeight01");
    myFlyweightFactory.create(&myFlyweightFactory, "myFlyWeight02");
    myFlyweightFactory.create(&myFlyweightFactory, "myFlyWeight03");
    myFlyweightFactory.create(&myFlyweightFactory, "myFlyWeight04");
    myFlyweightFactory.create(&myFlyweightFactory, "myFlyWeight05");
    myFlyweightFactory.create(&myFlyweightFactory, "myFlyWeight06");
    myFlyweightFactory.create(&myFlyweightFactory, "myFlyWeight07");
    myFlyweightFactory.create(&myFlyweightFactory, "myFlyWeight08");
    myFlyweightFactory.create(&myFlyweightFactory, "myFlyWeight09");
    myFlyweightFactory.create(&myFlyweightFactory, "myFlyWeight10"); /* Cannot
allocate flyweight in pool!!! */

    FLYWEIGHT *myFlyWeight = myFlyweightFactory.create(&myFlyweightFactory,
                                                       "myFlyWeight10"); /* Cannot allocate flyweight in pool!!! */

    myFlyWeight->operation(myFlyWeight->key, &extrinsic_state); /* The flyweight
information : None FlyWeight, Hello, My FlyWeight!!! */

    myFlyWeight = myFlyweightFactory.getFlyWeight(&myFlyweightFactory,
                                                "myFlyWeight09");
    myFlyWeight->operation(myFlyWeight->key, &extrinsic_state);

    return EXIT_SUCCESS;
}

```

예제는 여러개의 플라이 웨이트들을 생성한 후, 특정 플라이 웨이트를 얻어서 일을 시키는 것을 보여준다. 외부 상태 정보인 "extrinsic_state"를 전달해서 메시지를 출력하도록 했다. 결과는 아래와 같다. 생성된 플라이 웨이트에 특정 연산을 수행하는 것은 해당 플라이 웨이트(경량화 된 자료구조)에 연결된 원본 자료구조에 접근하는 것과 같이 생각해 볼 수 있다.

한정된 풀에서 자료구조를 생산하는 것은 모든 것을 한번에 보여줄 필요가 없이 보고 싶은 부분에 해당하는 것들로 한정 했다고 생각할 수 있다. 예를 들어, 한번에 전체 자료에 대한 것을 보여주기보다, 사용자의 입력에 따라 출력될 공간에 필요한 만큼씩 보여주는 것으로 실무에서 활용할 수 있다.

[결과]

Fly Weight Pattern Example 01

Flyweight created: 0

Flyweight created: 1

Flyweight created: 2

Flyweight created: 3

Flyweight created: 4

Flyweight created: 5

Flyweight created: 6

Flyweight created: 7

Flyweight created: 8

Flyweight created: 9

Cannot allocate flyweight in pool!!!

Cannot allocate flyweight in pool!!!

The flyweight information : None FlyWeight, Hello, My FlyWeight!!!

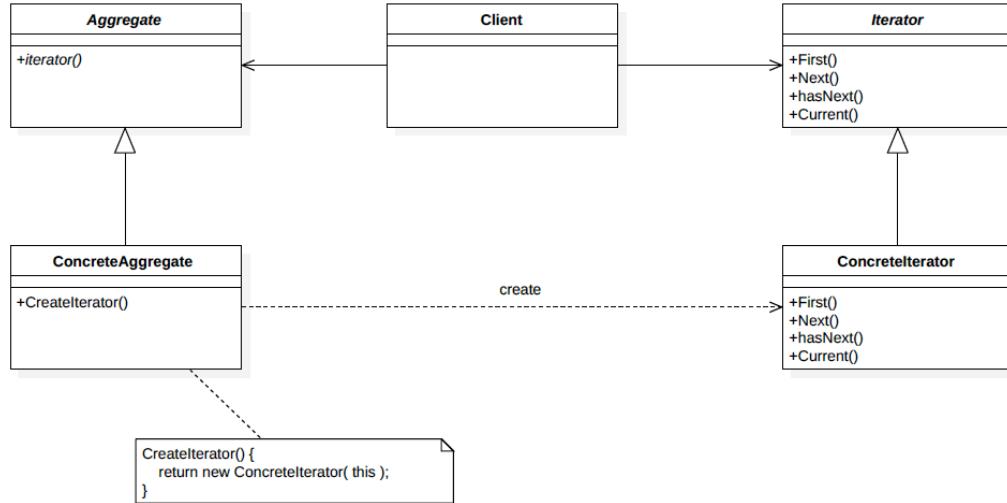
Found same flyweight!!!

The flyweight information : myFlyWeight09, Hello, My FlyWeight!!!

[반복자 패턴(Iterator Pattern)을 이용한 개별 원소 다루기]

반복자는 자료구조의 내부에 관련된 원소들을 저장하고 차례로 연산을 적용할 경우에 사용할 수 있다. 이 때 내부에 저장되는 형식에 대해서는 외부에서 모르며(감춰지며), 연산을 적용할 원소들을 특정 조건을 만족하는 것으로 한정하기 위해서 필터(Filter)를 명시할 수도 있다.

반복해서 특정 조건을 만족하는 원소들을 하나씩 찾을 수 있다고 해서 반복자(Iterator)라고 하며, 주로 루프(Loop)와 같은 순환문과 함께 자료구조에 포함된 모든 원소 들에 특정 연산을 수행하기 위해서 사용된다.



클라이언트는 원소들의 모음인 반복자를 생성하는 인터페이스를 알고 있으며, 반복자의 생성은 해당 인터페이스를 제공하는 반복자 생성자("ConcreteAggregate")를 통해서 이루어진다. 생성된 반복자를 이용해서 반복자 내부에 있는 첫번째 원소를 구하거나("First()"), 다음 원소 찾기("Next()"), 다음 원소가 있는지 확인하기("HasNext()"), 현재 접근하는 원소 구하기("Current()")등을 요청할 수 있다.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <stdbool.h>

typedef struct element ELEMENT;

struct element {
    unsigned int value;
    ELEMENT *next;
    ELEMENT *prev;
    unsigned int (*getValue)(ELEMENT *element);
};

unsigned int getValue(ELEMENT *element) {
    return element->value;
}
```

저장되는 원소들은 각각의 원소들을 연결할 수 있는 구조를 필요로 한다. 여기서는 그것을 만들기 위해 서 양방향 링크를 만들어 주었다("next, prev"). 각각의 원소들은 자신의 내부 자료에 대한 접근을 어떻게 할 수 있는지를 알려주는 인터페이스를 가진다("getValue()").

```
typedef struct iterator ITERATOR;

struct iterator {
    ELEMENT *head;
    ELEMENT *current;
    void (*addElement)(ITERATOR *iter, ELEMENT *element);
    ELEMENT *(*getNext)(ITERATOR *iter);
    bool (*hasNext)(ITERATOR *iter);
    void (*reset)(ITERATOR *iter);
};
```

반복자의 내부는 저장된 첫 번째 원소를 가리키기 위한 "head"와 현재 어떤 원소를 보고 있는지를 보여 주는 "current"를 가지고 있다. 그리고, 원소를 추가하고("addElement()"), 다음 원소를 구하는 ("getNext()") 함수를 기본적으로 가진다. 추가적으로 "current"가 가리키는 원소 다음에도 값을 가지는 원소가 있는지 확인하는 함수("hasNext()")와 "current"를 "head"로 위치를 변경해주는 "reset()" 함수를 가진다.

```
void addElement(ITERATOR *iter, ELEMENT *element) {
    ELEMENT *temp;
    if (iter->head == NULL) {
        iter->head = element;
        iter->current = iter->head;
        element->prev = NULL;
        element->next = NULL;
        return;
    }
    temp = iter->head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = element;
    element->next = NULL;
```

```

element->prev = temp;
return;
}

```

원소를 추가하는 것은 비어있을 경우와 그렇지 않을 경우로 나누어서 처리했다. 빈 경우에는 그냥 "head"에 연결시켜주고, 그렇지 않을 경우에는 제일 마지막에 추가하는 원소를 넣어주도록 했다. 추가 후에는 양방향 연결을 위한 처리로 마무리 지었다.

```

ELEMENT *getNext(ITERATOR *iter) {
    ELEMENT *temp = NULL;

    if ((temp = iter->current) != NULL) {
        iter->current = iter->current->next;
    }
    return temp;
}

bool hasNext(ITERATOR *iter) {
    printf("Here I am!!!\n");
    if (iter == NULL) {
        return false;
    }
    if (iter->current == NULL) {
        return false;
    }
    if (iter->current->next == NULL) {
        return false;
    }
    return true;
}

void reset(ITERATOR *iter) {
    iter->current = iter->head;
    return;
}

```

다음 원소를 구하는("getNext()")함수는 "current"에서 원소를 찾아내고, 다음 위치로 옮겨주는 것으로 구현했다. 다음 원소가 있는지를 파악하는("hasNext()")함수는 "current"의 다음을("next") 확인하는 방법으로 구현했다. "Reset()"함수는 "head"의 위치로 "current"를 변경하는 것으로 간단히 구현되었다.

```

#define LIMIT 300
ELEMENT *getNextMoreThanX(ITERATOR *iter) {
    ELEMENT *temp = NULL;

    while (((temp = iter->current) != NULL) && (iter->current->value < LIMIT)) {
        iter->current = iter->current->next;
    }
    if (temp == NULL)
    {

```

```

        return temp;
    }
    iter->current = iter->current->next;
    return temp;
}

```

"getNextMoreThanX()"함수는 주어진 특정한 값을 필터로 사용해서 다음 원소를 구해오는 것을 보여주기 위해서 구현했다. 즉, 필터 조건을 변경 함으로써, 저장된 각각의 원소를 찾아내는 방법을 변경할 수 있다.

```

int main(void) {
    puts("Iterator Pattern Example 01"); /* prints Iterator Pattern Example 01 */
    ELEMENT *element;
    ITERATOR iterator = { NULL, NULL, addElement, getNext, hasNext, reset };
    ELEMENT element01 = { 100, NULL, NULL, getValue };
    ELEMENT element02 = { 200, NULL, NULL, getValue };
    ELEMENT element03 = { 300, NULL, NULL, getValue };
    ELEMENT element04 = { 400, NULL, NULL, getValue };
    ELEMENT element05 = { 500, NULL, NULL, getValue };

    iterator.addElement(&iterator, &element01);
    iterator.addElement(&iterator, &element02);
    iterator.addElement(&iterator, &element03);
    iterator.addElement(&iterator, &element04);
    iterator.addElement(&iterator, &element05);

    /* Print all element in iterator */
    while ((element = iterator.getNext(&iterator)) != NULL) {
        printf("The element value : %d\n", element->getValue(element));
    }

    /* Reset iterator and set filter to get the element */
    printf("\n");
    iterator.reset(&iterator);
    iterator.getNext = getNextMoreThanX;
    printf("===\n");
    while ((element = iterator.getNext(&iterator)) != NULL) {
        printf("The element value : %d\n", element->getValue(element));
    }

    /* Has next test */
    iterator.reset(&iterator);
    if (iterator.hasNext(&iterator)) {
        printf("There are more elements in iterator!!!\n");
    }
    printf("Done!!!!\n");

    return EXIT_SUCCESS;
}

```

예제는 구현상의 편의를 위해서 정적으로 원소들을 만들어서 반복자에 추가해 주었다. 다양한 반복자의 연산이 반복문과 함께 제대로 사용될 수 있는지를 보기 위해서 작성했다. 필터에 대한 것도 추가되었다. 필터는 앞에서 설명했듯이 특정 조건을 만족하는 원소만 찾는데 활용할 수 있다.

반복자는 내부에 구현된 자료구조를 외부에서 반복적으로 하나씩 불러내서 사용할 수 있는 방법을 자료구조에 의존하지 않는 독립적인 방법으로 제공한다. 따라서, 사용하는 측과 실제 자료구조를 구현하는 측이 독립적으로 변경될 수 있다는 것이 장점이다.

[결과]

Iterator Pattern Example 01

The element value : 100

The element value : 200

The element value : 300

The element value : 400

The element value : 500

====

The element value : 300

The element value : 400

The element value : 500

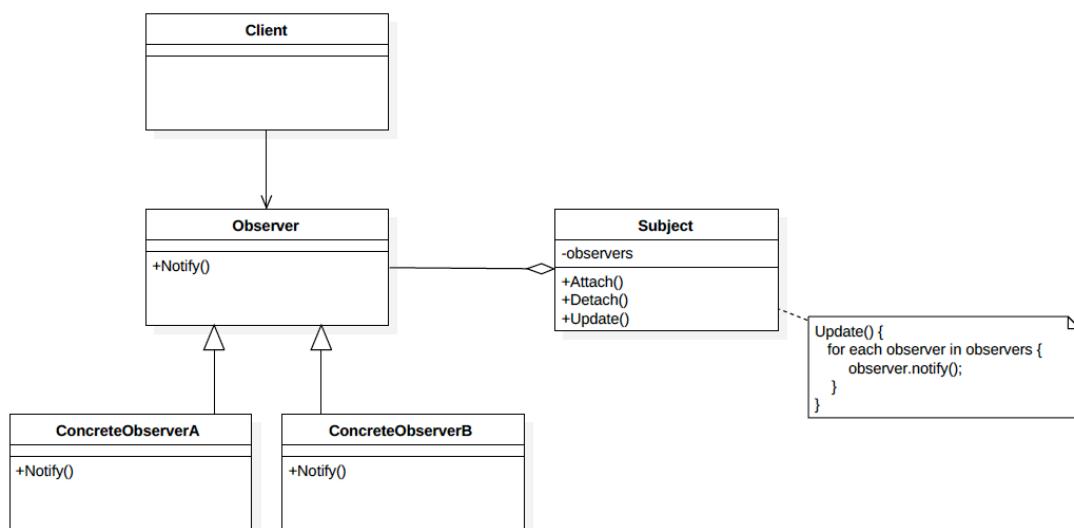
Here I am!!!

There are more elements in iterator!!!

Done!!!

[관찰자 패턴(Observer Pattern)을 이용한 변경 통보받기]

관찰자 패턴은 특정 이벤트가 발생했을 때, 상태의 변화를 알려달라고 요청할 때 사용할 수 있다. 즉, 상태 변화가 발생한 경우에만 깨워달라는 목적으로 사용한다. 예를 들어, 주식값의 변경이 발생하거나 새로운 출판물이 있는 경우에만 특정 일을 실행하고 싶을 때 이용할 수 있다.



관찰자는 관찰하고 있는 주제(Subject)가 있으며, 그 주제에 대한 변화 감지를 알려달라는 요청을 할 수 있어야 한다. 주제의 관점에서는 각각의 등록된 관찰자를 새로운 변화가 있을 때 알려주어야 ("Update()") 할 책임을 가진다. 관찰자는 알림을 받기 위해서 자신에게 알려줄 때 사용해야 할 인터페이스("Notify()")를 주제에 미리 등록("Attach()")해야 한다. 더 이상 알림을 받고 싶지 않을 때는

“Delete()”를 호출하도록 한다. 관찰자는 하나의 주제에 대해서 다양하게 있으며, 클라이언트는 각각의 관찰자를 만들어서 주제에 등록할 수 있다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct observer OBSERVER;

struct observer {
    char *key;
    char *message;
    void (*notify)(OBSERVER *observer);
};

void notify(OBSERVER *observer) {
    printf("The Observer Key : %s, The Message : %s\n", observer->key,
           observer->message);
    return;
}
```

관찰자는 자신을 구별하기 위한 "key"를 가지고 있으며, 특정한 이벤트가 발생했을 때 받고자 하는 메시지를 정의해 둘 수 있다. 그리고, 자신을 깨우기 위해서 호출되어야 하는 함수도 정의하고 있다 ("notify()").

```
#define MAX_OBSERVER_NUMBER 10 /* 최대 관찰자의 개수 */
typedef struct observer_collection OBSERVER_COLLECTION; /* 관찰자 모음의 자료구조 */

struct observer_collection {
    char *subject;
    OBSERVER *observer[MAX_OBSERVER_NUMBER];
    void (*addObserver)(OBSERVER_COLLECTION *obsc, OBSERVER *obs);
    void (*removeObserver)(OBSERVER_COLLECTION *obsc, OBSERVER *obs);
    void (*update)(OBSERVER_COLLECTION *obsc);
};
```

주제의 측면에서는 관찰자들의 모음을 관리할 수 있어야 한다. 이름을 "Collection"으로 정한 이유가 거기에 있다. 그리고, 관찰자를 추가하는 것("addObserver()")과 제거하는 것("removeObserver()")도 가질 수 있으며, 관찰자에게 변화가 일어났다는 것을 알리기 위한 인터페이스로 "update()"를 가진다.

```
void addObserver(OBSERVER_COLLECTION *obsc, OBSERVER *obs) {
    /* Find same observer in the collection */
    for (int i = 0; (i < MAX_OBSERVER_NUMBER) && (obsc->observer[i] != NULL);
         i++) {
        if (strncmp(obsc->observer[i]->key, obs->key, strlen(obs->key) + 1)
            == 0) {
            printf("Found same observer!!!\n");
            return;
        }
    }
}
```

```

/* Find empty slot */
int i = 0;
while ((obsc->observer[i] != NULL) && (i < MAX_OBSERVER_NUMBER)) {
    i++;
}
if (i == MAX_OBSERVER_NUMBER) {
    printf("Cannot add more observer!!!\n");
    return;
}

/* Add observer */
printf("Add Slot NUmber : %d\n", i);
obsc->observer[i] = obs;
return;
}

```

관찰자를 추가하는 것은 동일한 관찰자가 있는지 확인하고, 없을 때 빈 곳을 찾아서 넣어준다. 동일한 관찰자의 추가를 막고 한정된 개수의 관찰자만 유지하기 위한 것이다. 필요하다면 동적으로 변하는 자료구조를 사용해도 상관없다.

```

void removeObserver(OBSERVER_COLLECTION *obsc, OBSERVER *obs) {
    int i;

    /* Find observer */
    for (i = 0; i < MAX_OBSERVER_NUMBER; i++) {
        if ((obsc->observer[i] != NULL)
            && (strncmp(obsc->observer[i]->key, obs->key,
                         strlen(obs->key) + 1) == 0)) {
            break;
        }
    }
    /* Not found */
    if (i == MAX_OBSERVER_NUMBER) {
        printf("Cannot find observer!!!\n");
        return;
    }
    /* Remove observer */
    printf("Remove Slot Number : %d\n", i);
    obs->observer[i] = NULL;
    return;
}

```

제거는 제거하려는 관찰자와 동일한 관찰자가 있는지 찾는다. 만약, 존재한다면 간단히 빈 것으로 처리해둔다. 삭제 후에는 더 이상 주제에 관련된 업데이트된 정보는 전달되지 않을 것이다.

```

void sendNotification(OBSERVER_COLLECTION *obsc) {
    OBSERVER *temp;
    for (int i = 0; i < MAX_OBSERVER_NUMBER; i++) {
        temp = obsc->observer[i];
        if (temp != NULL) {
            if (temp->notify != NULL) {

```

```

        temp->notify(temp);
    }
}
return;
}

```

관찰자에게 알림을 주기 위해서 관찰자가 설정한 인터페이스를 그대로 이용한다("notify()"). 즉, 주제의 관점에서는 관찰자가 어떤 식으로 알림을 받아서 처리 하는지는 관심이 아니며, 관찰자의 책임인 것이다. 따라서, 관찰자마다 별도의 처리가 필요하면, "notify()" 인터페이스를 재정의 하면 될 것이다.

```

OBSERVER observer01 = { "Observer 01", "I am the observer 01", notify };
OBSERVER observer02 = { "Observer 02", "I am the observer 02", notify };
OBSERVER observer03 = { "Observer 03", "I am the observer 03", notify };
OBSERVER observer04 = { "Observer 04", "I am the observer 04", notify };
OBSERVER observer05 = { "Observer 05", "I am the observer 05", notify };

OBSERVER_COLLECTION observer_collection = { "My Subject", { }, addObserver,
                                           removeObserver, sendNotification };

int main(void) {
    puts("Observer Pattern Example 01"); /* prints Observer Pattern Example 01 */

    observer_collection.addObserver(&observer_collection, &observer01);
    observer_collection.addObserver(&observer_collection, &observer02);
    observer_collection.addObserver(&observer_collection, &observer03);
    observer_collection.addObserver(&observer_collection, &observer04);
    observer_collection.addObserver(&observer_collection, &observer05);

    observer_collection.removeObserver(&observer_collection, &observer02);
    observer_collection.removeObserver(&observer_collection, &observer01);
    observer_collection.removeObserver(&observer_collection, &observer04);
    observer_collection.removeObserver(&observer_collection, &observer03);
    observer_collection.removeObserver(&observer_collection, &observer05);

    observer_collection.addObserver(&observer_collection, &observer01);
    observer_collection.addObserver(&observer_collection, &observer02);
    observer_collection.addObserver(&observer_collection, &observer03);
    observer_collection.addObserver(&observer_collection, &observer04);
    observer_collection.addObserver(&observer_collection, &observer05);
    observer_collection.update(&observer_collection);
    return EXIT_SUCCESS;
}

```

구현의 편의를 위해서 정적으로 정의된 관찰자들과 주제를 생성했다. 추가와 삭제가 제대로 되는지를 확인하기 위해서 테스트를 했으며, 새로운 상태를 업데이트하는 예를 만들었다. 예제를 실행한 결과는 아래와 같다.

[결과]

Observer Pattern Example 01

Add Slot NUmber : 0

```

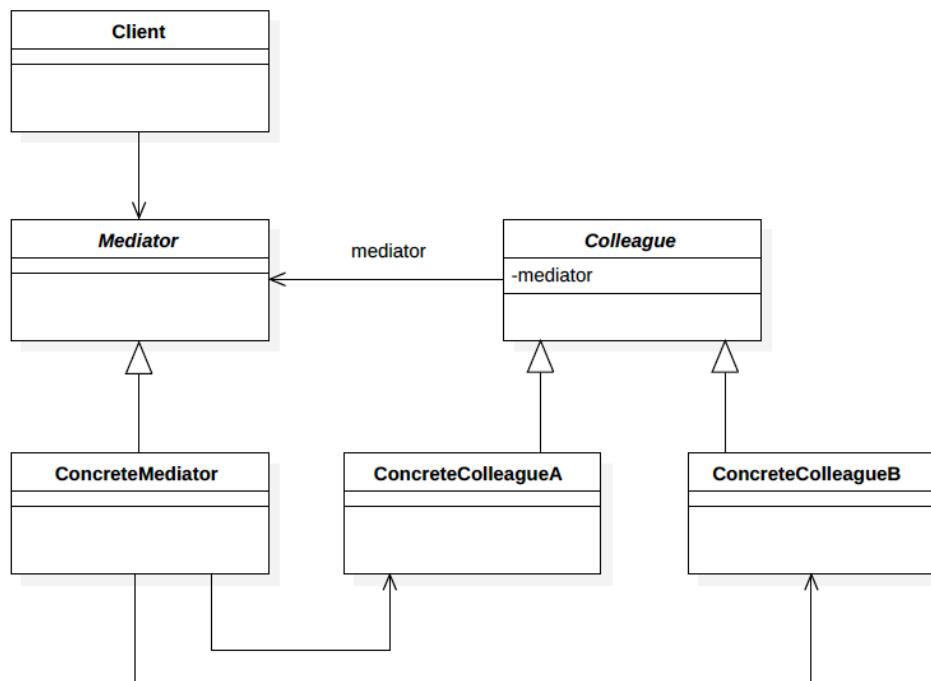
Add Slot NUmber : 1
Add Slot NUmber : 2
Add Slot NUmber : 3
Add Slot NUmber : 4
Remove Slot Number : 1
Remove Slot Number : 0
Remove Slot Number : 3
Remove Slot Number : 2
Remove Slot Number : 4
Add Slot NUmber : 0
Add Slot NUmber : 1
Add Slot NUmber : 2
Add Slot NUmber : 3
Add Slot NUmber : 4
The Observer Key : Observer 01, The Message : I am the observer 01
The Observer Key : Observer 02, The Message : I am the observer 02
The Observer Key : Observer 03, The Message : I am the observer 03
The Observer Key : Observer 04, The Message : I am the observer 04
The Observer Key : Observer 05, The Message : I am the observer 05

```

[중재자 패턴(Mediator Pattern)을 이용한 통신의 구현]

중재자 패턴은 다른 자료구조들을 위한 중재역할을 하기 위해서 사용한다. 즉, 서로 상대방에 대해서 모르는 자료구조들간에 중재역할을 한다. 여기서는 그것을 통신의 역할을 하는 것으로 예를 들어보도록 하겠다. 서로 통신을 나누는 것들은 상대방에 대해서는 전혀 모르고 있지만, 중재자에 대해서는 알고 있어야 한다. 즉, 중재자에 중재 역할을 맡기기 위해서는 어떻게 사용하는지 정도는 알아야 한다는 것이다.

중재자는 자신이 중재를 관여하는 것들에는 어떤 것들이 있는지를 알아야 하며, 중재를 맡기는 측에서는 중재자가 누구인지를 알아야 한다. 따라서, 중재자와 양방향으로 연결이 필요하다는 것을 알 수 있을 것이다.



클라이언트는 중재자("Mediator")를 통해서 다른 협력자("Colleague")를 사용할 수 있으며, 마찬가지로 각각의 협력자도 항상 중재자("mediator")를 통해서 다른 협력자나 클라이언트를 찾을 수 있다. 중재자나 협력자는 정의된 자료구조와 인터페이스를 따른다면, 협력의 파트너 관계로 언제든 추가될 수 있다. 물론, 이때도 마찬가지로 항상 중재자를 통해서 접근해야만 사용할 수 있다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct message {
    char *message;
} MESSAGE;

typedef struct mediator MEDIATOR; /* 중재자를 미리 정의 */
typedef struct colleague COLLEAGUE;

struct colleague {
    char *key;
    MEDIATOR *mediator;      /* 중재자에 대한 링크 */
    void (*send)(COLLEAGUE *colleague, MESSAGE *msg);
    void (*receive)(COLLEAGUE *colleague, MESSAGE *msg);
};

#define MAX_COLLEAGUE_NUMBER 5
struct mediator {
    COLLEAGUE *colleague[MAX_COLLEAGUE_NUMBER];
    void (*addColleague)(MEDIATOR *mediator, COLLEAGUE *colleague);
    void (*removeColleague)(MEDIATOR *mediator, COLLEAGUE *colleague);
    void (*send)(MEDIATOR *mediator, COLLEAGUE *colleague, MESSAGE *msg);
};
```

통신 상에 전달되는 것을 메시지로 표현했다. 실제 구현에서는 이런 메시지에 각종 필드를 더 추가하면 될 것이다. 중재자와 중재에 참여하는 측이 서로의 타입에 대해서는 알아야 하기에 타입의 선언이 구조체 정의보다 먼저 나온다. 중재에 참여하는 측은 중재자에게 메시지를 전달하는 요청("send()")과 중재자 측에서 메시지의 수신을 받기 위한 인터페이스("receive()")를 정해주어야 한다.

중재자 측에서는 중재에 참여하는 것들의 리스트를 관리하기 위해서 배열을 사용했다. 새로운 배열의 요소를 추가하기 위해서 "addColleague()"과, 삭제를 위한 "removeColleague()"을 정의해 주었다. 중재를 요청하는 측에서 메시지를 보내기 위해서 중재자를 호출하는 인터페이스도 정의했다("send()").

```
void sendColleague(COLLEAGUE *colleague, MESSAGE *msg) {
    printf("Colleague Send Requested!!!\n");
    if (colleague->mediator == NULL) {
        printf("Cannot send with NULL mediator!!!\n");
        return;
    }
    colleague->mediator->send(colleague->mediator, colleague, msg);
    return;
}

void receiveColleague(COLLEAGUE *colleague, MESSAGE *msg) {
    printf("The %s received : %s\n", colleague->key, msg->message);
```

```

    return;
}

```

중재자에 메시지 중재 요청을 보내는 것은 단순히 중재자의 "send()"를 호출하면 된다. 중재자는 중재요청에 대해서 자신이 관리하는 것들에 정의된 모든 "receive()"를 보내려는 메시지와 함께 대신 호출해줄 것이다.

전달받은 메시지는 단순히 누가 받았는지를 확인하고, 메시지의 내용을 화면상에 표시하는 것으로 끝난다. 필요하다면, 여기에 추가적인 메시지를 처리하는 부분을 구현해도 될 것이다.

```

void addColleague(MEDIATOR *mediator, COLLEAGUE *colleague) {
    int i;

    /*Find colleague is already there */
    for (i = 0; i < MAX_COLLEAGUE_NUMBER; i++) {
        if (mediator->colleague[i] == NULL) {
            continue;
        }
        if (strncmp(mediator->colleague[i]->key, colleague->key,
                    strlen(colleague->key) + 1) == 0) {
            break;
        }
    }
    if (i != MAX_COLLEAGUE_NUMBER) {
        printf("We found same colleague : %d\n", i);
        return;
    }

    /* Find empty slot */
    for (i = 0; i < MAX_COLLEAGUE_NUMBER; i++) {
        if (mediator->colleague[i] == NULL) {
            break;
        }
    }
    if (i == MAX_COLLEAGUE_NUMBER) {
        printf("Cannot find empty slot!!!\n");
        return;
    }

    /* Add colleague */
    mediator->colleague[i] = colleague;
    colleague->mediator = mediator;
    return;
}

```

중재를 요청하는 것들을 간단히 관리하기 위해서 배열을 사용했다. 먼저 이미 등록된 것이 있는지 찾고 없다면 추가해 주도록 한다. 만약, 더 이상 추가할 공간이 없다면 오류를 표시하고 복귀하게 된다. 추가할 빈 공간이 있다면, 찾은 빈 곳에 중재 요청자를 넣어두도록 한다. 한정된 개수가 많다면, 포인터를 이용해서 연결 리스트로 구현해도 될 것이다. 혹은, 반복자(Iterator)와 같은 것을 사용해서 구현할 수도 있다.

```

void removeColleague(MEDIATOR *mediator, COLLEAGUE *colleague) {
    int i;
    COLLEAGUE *temp;

    /* Find colleague */
    for (i = 0; i < MAX_COLLEAGUE_NUMBER; i++) {
        temp = mediator->colleague[i];
        /* Empty? */
        if (temp == NULL) {
            continue;
        }
        /* Remove colleague */
        if (strncmp(temp->key, colleague->key, strlen(colleague->key) + 1)
            == 0) {
            printf("We found the colleague : %s\n", colleague->key);
            mediator->colleague[i] = NULL;
            return;
        }
    }
    printf("Cannot find the colleague : %s\n", colleague->key);
    return;
}

```

중재 요청자를 제거하기 위해서는 검색해서 같은 것이 있는지를 찾아야 할 것이다. 찾았다면 해당 배열의 원소를 빈 것으로 만들어준다. 찾을 수 없다면 오류 메시지를 출력하고 복귀할 것이다.

```

void mediatorSend(MEDIATOR *mediator, COLLEAGUE *colleague, MESSAGE *msg) {
    COLLEAGUE *temp;

    for (int i = 0; i < MAX_COLLEAGUE_NUMBER; i++) {
        temp = mediator->colleague[i];
        /* Empty? */
        if (temp == NULL) {
            continue;
        }
        /* Skip same colleague */
        if (temp == colleague) {
            continue;
        }
        temp->receive(temp, msg);
    }
    return;
}

```

중재자의 중재 요청의 전달은 중재 요청자들이 가지고 있는 "receive()"함수를 호출하는 것으로 처리된다. 즉, 배열에 저장된 중재 요청자들을 하나씩 둘러보면서, 보낸 것과 같지 않은 중재 요청자들의 "receive()"를 각각 호출해 준다. 만약, 특정한 중재 요청자에게 메시지를 전달하고자 한다면, 이곳에서 특정 중재 요청자를 찾을 수 있도록 해야할 것이다.

```

COLLEAGUE colleague00 = { "Colleague 00", NULL, sendColleague, receiveColleague };
COLLEAGUE colleague01 = { "Colleague 01", NULL, sendColleague, receiveColleague };

```

```

COLLEAGUE colleague02 = { "Colleague 02", NULL, sendColleague, receiveColleague };
COLLEAGUE colleague03 = { "Colleague 03", NULL, sendColleague, receiveColleague };
COLLEAGUE colleague04 = { "Colleague 04", NULL, sendColleague, receiveColleague };
COLLEAGUE colleague05 = { "Colleague 05", NULL, sendColleague, receiveColleague };

MEDIATOR mediator = { { }, addColleague, removeColleague, mediatorSend };

int main(void) {
    puts("Mediator Pattern Example 01");

    mediator.addColleague(&mediator, &colleague00);
    mediator.addColleague(&mediator, &colleague01);
    mediator.addColleague(&mediator, &colleague02);
    mediator.addColleague(&mediator, &colleague03);
    mediator.addColleague(&mediator, &colleague04);
    mediator.addColleague(&mediator, &colleague05); /* Cannot find empty slot!!! */

    MESSAGE msg = { "Hello, World!!!" };

    mediator.removeColleague(&mediator, &colleague05);/* Cannot find the colleague :
Colleague 05 */
    mediator.removeColleague(&mediator, &colleague03);/* We found the colleague :
Colleague 03 */
    colleague00.send(&colleague00, &msg);

    mediator.addColleague(&mediator, &colleague03);
    colleague00.send(&colleague00, &msg);
    return EXIT_SUCCESS;
}

```

구현의 편의를 위해서 정적으로 생성된 중재 요청자와 중재자를 만들어 주었다. 한정된 수 이상의 중재 요청자를 추가했을 때 발생하는 오류와 삭제 하려는 중재 요청자가 없을 때 어떤 일이 발생하는지도 알려준다. 중재 요청자들이 중재 요청을 보냈을 때 어떤 반응을 보여줄지도 예도 만들었다.

중재자를 기준으로 중재를 요청하는 측과 중재 요청의 대상이 분리되어 관리될 수 있다는 점에서 의존성을 낮춰준다고 볼 수 있다. 즉, 중재자를 기준으로 중재 요청자들 각각이 독립적으로 변경될 수 있다. 또한, 새로운 중재 요청자를 추가해서 동일한 메시지에 대해서 다른 처리를 구현할 가능성도 열어두고 있다는 점에서 확장성이 용이한 측면도 가지고 있다.

[결과]

```

Mediator Pattern Example 01
Cannot find empty slot!!!
Cannot find the colleague : Colleague 05
We found the colleague : Colleague 03
Colleague Send Requested!!!
The Colleague 01 received : Hello, World!!!
The Colleague 02 received : Hello, World!!!
The Colleague 04 received : Hello, World!!!
Colleague Send Requested!!!
The Colleague 01 received : Hello, World!!!
The Colleague 02 received : Hello, World!!!

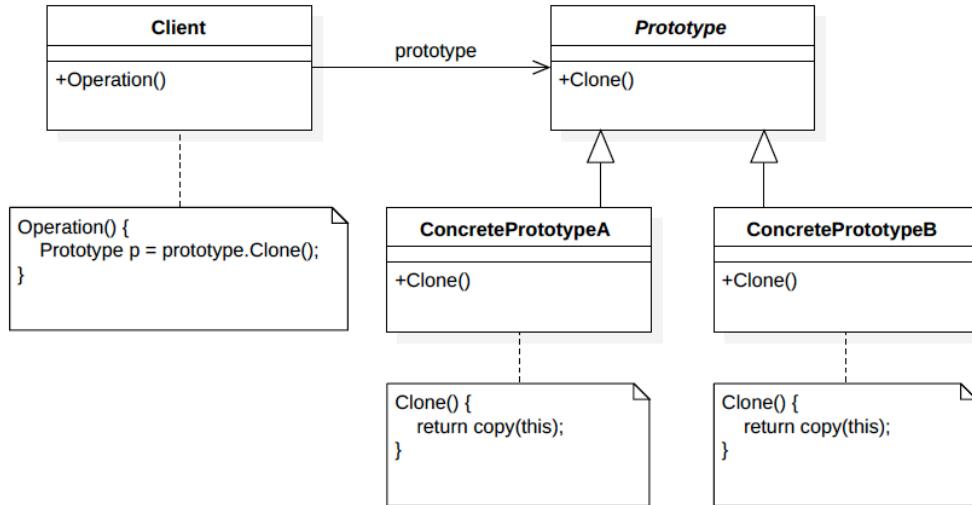
```

The Colleague 03 received : Hello, World!!!

The Colleague 04 received : Hello, World!!!

[프로토타입 패턴(Prototype Pattern)을 이용한 복제(Clone) 생성]

프로토타입 패턴은 간단히 원하는 자료형을 만들어내고자 할 때 사용할 수 있다. 즉, 이미 정해져 있는 타입에 맞게 그것을 복제해서 새로운 자료구조를 생성할 때 사용할 수 있다. 각각의 프로토타입은 자신의 고유한 연산을 재정의 할 수 있으며, 그로부터 생성될 모든 프로토타입 들에 대해서 동일한 연산을 적용할 수 있도록 만들어 준다.



클라이언트의 연산은 생성된 프로토타입("prototype")이 처리할 수 있으며, 프로토타입은 여러가지로 정의 될 수 있다("ConcretePrototypeA, ConcretePrototypeB"). 각각의 프로토타입은 자신과 동일한 자료구조를 생성하기 위한 인터페이스를 정의하고 있으며("Clone()"), 이를 통해서 클라이언트의 생성 요청을 처리할 수 있다.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct prototype PROTOTYPE;
struct prototype {
    char *key;
    PROTOTYPE *(*clone)(PROTOTYPE *proto);
    void (*do_something)(PROTOTYPE *prototype);
};
  
```

예제는 특정한 키(Key) 값을 가지는 프로토타입을 만들기 위해서 자료형을 정의했다. 이때, 각각의 프로토타입을 생성하기 위해서는 복제를 할 수 있는 인터페이스를 가져야하며, 여기서는 "clone()"이 그와 같은 역할을 하고 있다. 나머지 함수는("do_something()") 생성된 프로토타입이 어떤 일을 할 수 있는지를 간단히 보여주기 위해서 사용했다.

```

static void do_something(PROTOTYPE *prototype) {
    printf("The name of prototype : %s\n", prototype->key);
}
  
```

```

static PROTOTYPE *clone(PROTOTYPE *proto) {
    PROTOTYPE *temp;
    if ((temp = (PROTOTYPE *) malloc(sizeof(PROTOTYPE))) == NULL) {
        printf("Cannot create prototype!!!\n");
        return NULL;
    }
    if ((temp->key = (char *) malloc(strlen(proto->key) + 1)) == NULL) {
        printf("Cannot create prototype key memory!!!\n");
        free(temp);
        return NULL;
    }
    strncpy(temp->key, proto->key, strlen(proto->key) + 1);
    temp->clone = proto->clone;
    temp->do_something = proto->do_something;
    return temp;
}

```

예제는 간단히 자신의 정보를 표시하는 것으로 하는 일을 구현했다. 복제(Clone)는 새로운 메모리를 할당받아서, 기존의 프로토타입을 복사하는 과정을 거치도록 했다. 따라서, 이 과정을 거치게 되면 동일한 역할을 하는 두 개의 자료구조가 만들어지게 된다.

여기서는 내부 함수는 동일하게 유지하고 있도록 만들었으나, 필요하다면 추가적인 파라미터를 명시하거나 별도의 인터페이스를 이용해서 새로 정해 줄 수 있을 것이다. 하지만, 사용하는 측에서는 기존의 인터페이스를 이용해서 계속 사용할 수 있도록 만들어 주어야 할 것이다.

```

int main(void) {
    puts("Prototype Pattern Example 01");
    PROTOTYPE proto = { "Prototype 01", clone, do_something};
    PROTOTYPE *myProto;

    proto.do_something( &proto );
    myProto = proto.clone( &proto );
    if ( myProto != NULL )
    {
        myProto->do_something( myProto );
    }

    return EXIT_SUCCESS;
}

```

프로토타입의 생성자 역할을 할 "proto"를 만들고, 어떤 역할을 하는지 호출해 보았다 ("do_something()"). 나중에 생성된 프로토타입의 복제와 비교해보기 위함이다.

프로토타입은 유사한 객체를 생성할 때 이미 알고 있는 자료형에 기반해서 생성하도록 만든다. 따라서, 기존에 이미 객체에 대한 사용법을 알고 있는 클라이언트 코드에서는 동일한 인터페이스를 통해서 새로 만들어진 객체를 사용할 수 있다. 즉, 새로운 자료구조라고 하더라도 유사한 방법으로 동일하게 사용할 수 있게 된다. 이를 통해서 유사하지만 새로운 기능을 추가하는 것이 쉽게 가능하다. 즉, 사용자 측의 코드는 변경할 필요가 없이, 유사하지만 새로운 기능을 추가할 수 있는 것이다.

[결과]

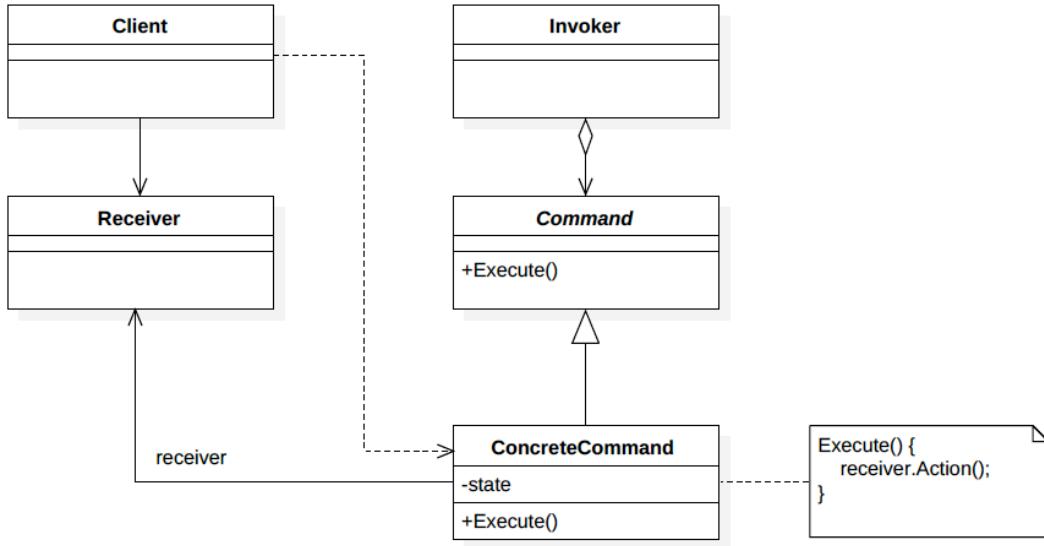
Prototype Pattern Example 01

The name of prototype : Prototype 01

The name of prototype : Prototype 01

[커맨드 패턴(Command Pattern)을 이용한 명령어 처리]

컨맨드(명령) 패턴은 커맨드 자체를 자료구조 형식으로 만들어서 처리를 위탁한다. 이렇게 만들어진 커맨드는 나중에 한번에 모아서 실행될 수 있으며, 역으로 원래의 상태를 되돌릴수 있는 로그를 남길 수도 있다. 따라서, 커맨드 들을 모아서 관리하고 커맨드를 실행하기 위한 자료구조도 포함된다.



클라이언트를 대상으로 여러가지 명령("Command")이 실행될 수 있으며, 각각의 명령("ConcreteCommand")은 "Invoker"에 의해서 보관 및 관리된다. 각각의 명령은 내부적으로 상태("state")를 가지며, 실행("Execute()")되기 위해서는 클라이언트가 알고 있는 "Receiver"의 인터페이스를 사용한다("Action()").

예제에서는 간단히 각각의 커맨드 타입을 정의하고, 커맨드를 관리하기 위한 자료구조를 제공하는 방법을 보도록 하겠다. 커맨드를 관리하는 자료구조는 커맨드의 실행과 추가, 삭제, 취소(Undo) 기능을 제공하도록 만들었다.

```

#include <stdio.h>
#include <stdlib.h>

typedef struct message MESSAGE;

struct message {
    char *message;
};

typedef enum {
    COMMAND_TYPE_A = 0, COMMAND_TYPE_B, COMMAND_TYPE_NONE
} COMMAND_TYPE;

typedef struct command COMMAND;

struct command {
    char *command;
}
  
```

```

COMMAND_TYPE type;
void (*execute)(COMMAND *command, MESSAGE *msg);
};

```

커맨드는 커맨드 타입으로 나눌 수 있도록 했으며("COMMAND_TYPE"), 각각의 커맨드는 이름과 어떻게 처리될 수 있는지를 자체 자료구조에 가질 수 있도록 만들었다. 즉, 커맨드를 관리하는 측에서는 커맨드를 어떻게 처리할지 모르더라도 커맨드 자료구조에서 제공된 인터페이스를 이용해서 커맨드를 실행 시킬 수 있다.

```

void execute(COMMAND *command, MESSAGE *msg) {
    switch (command->type) {
        case COMMAND_TYPE_A:
            printf("This is type A command : Name - %s, Message - %s\n",
                   command->command, msg->message);
            break;
        case COMMAND_TYPE_B:
            printf("This is type A command : Name - %s, Message - %s\n",
                   command->command, msg->message);
            break;
        default:
            printf("This is unknown type command : Name - %s, Message - %s\n",
                   command->command, msg->message);
            break;
    }
    return;
}

```

커맨드 자체는 타입에 따라, 전달된 메시지와 커맨드의 이름 등을 출력하도록 했다. 각각의 커맨드 별로 처리 방식을 따로 정할 수 있지만, 여기서는 구현을 쉽게하기 위해서 하나의 함수로 처리하도록 만들었다.

```

#define MAX_COMMAND_NUMBER 5
typedef struct manager MANAGER;
struct manager {
    COMMAND *commands[MAX_COMMAND_NUMBER];
    unsigned int command_count;
    void (*addCommand)(MANAGER *manager, COMMAND *command);
    void (*flushCommands)(MANAGER *manager);
    void (*runCommands)(MANAGER *manager, MESSAGE *msg);
    void (*undoCommand)(MANAGER *manager);
};

```

커맨드 관리자("MANAGER")는 전달받은 커맨드들을 저장하고("commands[]"), 현재 저장된 커맨드의 갯수("command_count"), 커맨드를 추가("addCommand()"), 비우기("flushCommand()"), 실행("runCommand()"), Undo 기능("undoCommand()")에 대한 인터페이스를 가진다.

```

void addCommand(MANAGER *manager, COMMAND *command) {
    /* Find empty slot */
    for (int i = 0; i < MAX_COMMAND_NUMBER; i++) {
        if (manager->commands[i] == NULL) {
            manager->commands[i] = command;

```

```

        manager->command_count++;
        return;
    }
}
printf("Cannot find empty slot in command list!!!\n");
return;
}

void flushCommands(MANGER *manager) {
    for (int i = 0; i < MAX_COMMAND_NUMBER; i++) {
        manager->commands[i] = NULL;
    }
    manager->command_count = 0;
    printf("Flush all commands done!!!\n");
    return;
}

void runCommands(MANGER *manager, MESSAGE *msg) {
    if (manager->command_count <= 0) {
        printf("There is no commands to be executed!!!\n");
        manager->command_count = 0;
        return;
    }
    for (int i = 0; i < MAX_COMMAND_NUMBER; i++) {
        if (manager->commands[i] != NULL) {
            manager->commands[i]->execute(manager->commands[i], msg);
            manager->commands[i] = NULL;
            manager->command_count--;
        }
    }
    printf("Run all commands done!!!\n");
    return;
}

void undoCommand(MANGER *manager) {
    if (manager->command_count <= 0) {
        printf("Cannot make undo for empty command list!!!\n");
        manager->command_count = 0;
        return;
    }
    manager->commands[manager->command_count - 1] = NULL;
    manager->command_count--;
    return;
}

```

커맨드를 추가하는 것은 단순히 커맨드의 포인터를 배열에 추가하는 것으로 처리했다. 커맨드를 비우는 것은 배열의 원소들을 모두 "NULL"로 만들고, 현재 저장된 커맨드의 수("command_count")를 "0"으로 만들었다. 커맨드를 실행하는 것은 저장된 각각의 커맨드 들을 불러 와서, 커맨드에서 제공하는 인터페이스("execute()")를 이용했다. 따라서, 커맨드 관리자는 커맨드 들을 실제로 어떻게 실행해야 할지를 알지 못한다. 전달 받은 커맨드를 "Undo"하는 기능은, 아직 실행되지 않은 커맨드 들을 가지고 있는 관

리자가 배열을 마지막에 전달받은 커맨드를 비우는 것으로 구현했다. 즉, 가장 마지막에 실행된 명령어를 하나 제거하는 것이다.

```
COMMAND command00 = { "Command 00", COMMAND_TYPE_A, execute };
COMMAND command01 = { "Command 01", COMMAND_TYPE_B, execute };
COMMAND command02 = { "Command 02", COMMAND_TYPE_NONE, execute };
COMMAND command03 = { "Command 03", COMMAND_TYPE_A, execute };
COMMAND command04 = { "Command 04", COMMAND_TYPE_B, execute };
COMMAND command05 = { "Command 05", COMMAND_TYPE_B, execute };
```

```
MANAGER manager =
{ {}, 0, addCommand, flushCommands, runCommands, undoCommand };
```

각각의 커맨드는 이름과 타입 만을 달리했고, 구현 편의상 처리 방식은 동일하게 적용하도록 만들었다. 만약, 각각의 커맨드 별로 다르게 처리 하려면, 커맨드 별 처리방식을 함수로 만들어서 넣어주면 될 것이다. 커맨드를 관리하는 매니저는 초기화를 간략하게 하기위해서 정적으로 정의 했으며, 정의된 함수들로 인터페이스를 채워주었다.

```
int main(void) {
    puts("Command Pattern Example 01"); /* prints Command Pattern Example 01 */
    MESSAGE message = { "Hello, World!!!" };

    manager.addCommand(&manager, &command00);
    manager.addCommand(&manager, &command01);
    manager.addCommand(&manager, &command02);
    manager.addCommand(&manager, &command03);
    manager.addCommand(&manager, &command04);
    manager.addCommand(&manager, &command05); /* Cannot find empty slot in
command list!!! */
    manager.flushCommands(&manager); /* Flush all commands done!!! */

    manager.addCommand(&manager, &command00);
    manager.addCommand(&manager, &command01);
    manager.addCommand(&manager, &command02);
    manager.addCommand(&manager, &command03);
    manager.addCommand(&manager, &command04);

    manager.runCommands(&manager, &message); /* Run all commands done!!! */
    manager.runCommands(&manager, &message); /* There is no commands to be
executed!!! */
    manager.addCommand(&manager, &command00);
    manager.addCommand(&manager, &command01);
    manager.addCommand(&manager, &command02);
    manager.addCommand(&manager, &command03);
    manager.addCommand(&manager, &command04);
    manager.undoCommand(&manager);
    manager.undoCommand(&manager);
    manager.runCommands(&manager, &message); /* Run all commands done!!! */

    return EXIT_SUCCESS;
}
```

예제는 정의된 커맨드 구조체 들을 추가하고 비우기, 커맨드를 실행하는 것 등을 보여준다. 커맨드 배열이 빈 경우에는 아무런 일을 하지 않는다. 커맨드 들을 채워서 “Undo”하는 과정도 보여주고 있다.

커맨드 패턴은 앞의 예제에서 보았듯이 “Undo”나 “Redo”기능을 구현하기 위해서 활용할 수 있다. 즉, 최종 저장 이후에 실행된 명령들을 보관하고 있다가, 실행을 취소하거나 취소된 명령을 다시 실행할 수 있도록 구현할 수 있다. 명령 자체와 명령에 관련된 데이터를 보관하고, 명령과 관련된 함수들을 연결해서 관리하는 방법으로 활용할 수 있을 것이다. 또한, 새로운 명령도 정의된 명령에 대한 자료구조와 인터페이스를 지킨다면 추가하는 것도 어렵지 않다는 것을 확인할 수 있다.

[결과]

Command Pattern Example 01

Cannot find empty slot in command list!!!

Flush all commands done!!!

This is type A command : Name - Command 00, Message - Hello, World!!!

This is type A command : Name - Command 01, Message - Hello, World!!!

This is unknown type command : Name - Command 02, Message - Hello, World!!!

This is type A command : Name - Command 03, Message - Hello, World!!!

This is type A command : Name - Command 04, Message - Hello, World!!!

Run all commands done!!!

There is no commands to be executed!!!

This is type A command : Name - Command 00, Message - Hello, World!!!

This is type A command : Name - Command 01, Message - Hello, World!!!

This is unknown type command : Name - Command 02, Message - Hello, World!!!

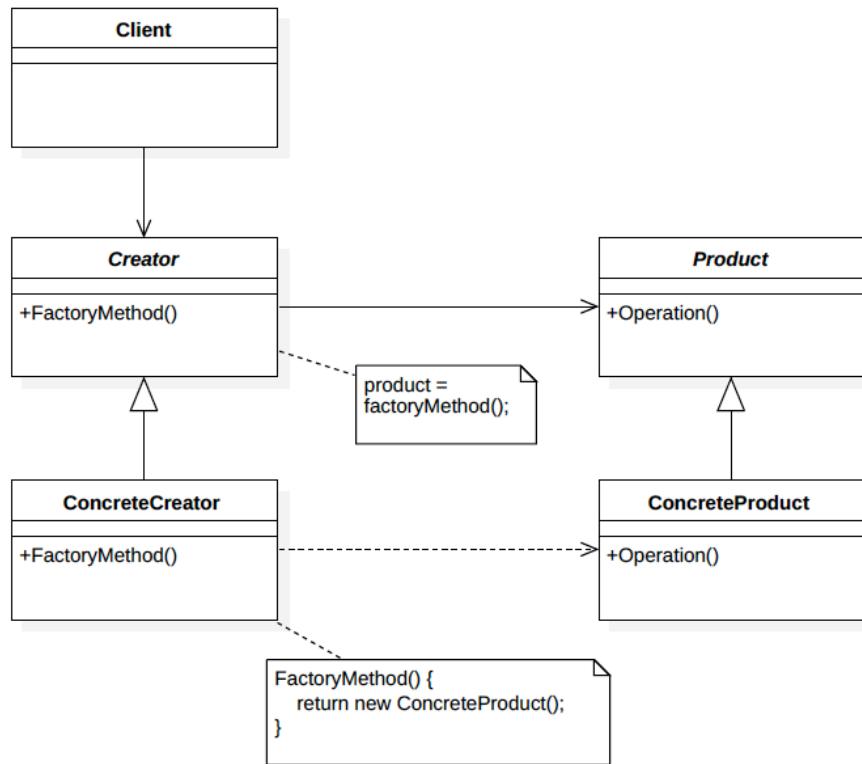
Run all commands done!!!

[팩토리 메소드 패턴(Factory Method Pattern)을 활용한 다양한 제품의 생산]

팩토리 메서드는 공장(팩토리)이 어떤 제품을 생산할지 구체적으로 알지 못할 경우에 사용할 수 있다. 구체적인 제품의 생산은 제품을 생산하는 과정에서 알려지게 되며, 팩토리 자체는 단지 외부에 제품을 만들어 보여주는 공장의 역할만 맡는다.

팩토리가 제품을 만들어내는 골격은 정의해 주지만, 그 골격을 채워주는 역할은 구체적인 제품의 생산을 담당하는 곳에서 담당한다. 따라서, 팩토리에 제품의 주문을 의뢰하는 측은 제품의 외관 골격만 볼 수 있고, 실제적인 제품의 생산은 실제 팩토리에서 담당한다.

팩토리 메서드 패턴도 잘 활용하면 프로그램의 분기 문제를 해결하는 효과적인 방법으로 사용할 수 있다. 즉, 생산하는 제품별로 다른 연산을 실행해야 할 경우, 실행해야 할 연산의 인터페이스를 정의한 후, 해당 인터페이스를 제공하는 함수를 생산해서 연결하는 방법으로 분기문을 사용하지 않고도 구현할 수 있도록 만들어 준다.



클라이언트는 제품을 만들 생성자("Creator")의 인터페이스를 알고 있으며, 실제 생산을 담당하는 것은 "ConcreteCreator"의 역할이다. 물론, 생성된 제품은 클라이언트가 원하는 일을 실행해 줄 것이다 ("Operation()"). 생성자의 역할은 단순히 클라이언트의 요구에 맞게 자신이 생산할 수 있는 제품을 만들어서 전달해 주면 된다.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct message {
    char *message;
} MESSAGE;

typedef enum {
    PRODUCT_TYPE_A = 0,
    PRODUCT_TYPE_B,
    PRODUCT_TYPE_C,
    PRODUCT_TYPE_NONE
} PRODUCT_TYPE;

typedef struct factory FACTORY, PRODUCT;

struct factory {
    char *name;
    PRODUCT *(*makeProduct)(PRODUCT_TYPE type);
    void (*do_something)(PRODUCT *product, MESSAGE *msg);
};

static void productDoSomething(PRODUCT *product, MESSAGE *msg) {
    // Implementation of do_something
}
    
```

```

        printf("The product name : %s, Message : %s\n", product->name, msg->message);
        return;
    }

static PRODUCT *nullMakeProduct(PRODUCT_TYPE type);
PRODUCT nullProduct = { "Null Product", nullMakeProduct, productDoSomething };

static PRODUCT *nullMakeProduct(PRODUCT_TYPE type) {
    return &nullProduct;
}

```

팩토리와 제품(Product)이 동일한 인터페이스와 자료구조를 사용하기에 같은 타입으로 정의했다. 나중에 더 추가적인 인터페이스가 필요하다면, 이것도 분리해서 정의할 수 있다. 제품이 할 수 있는 일을 정의하기 위해서 인터페이스로 "do_something()"을 만들었으며, 제품의 타입과 제품이 처리할 메시지의 자료구조(MESSAGE)도 선언해 주었다.

```

static PRODUCT *makeProduct(PRODUCT_TYPE type) {
    PRODUCT *product;
    char *typeAName = "Product A";
    char *typeBName = "Product B";
    char *typeCName = "Product C";
    char *productName = NULL;

    /* Allocate product structure */
    if ((product = (PRODUCT *) malloc(sizeof(PRODUCT))) == NULL) {
        printf("Cannot create product!!!\n");
        return &nullProduct;
    }

    /* Find product type and name */
    switch (type) {
        case PRODUCT_TYPE_A:
            productName = typeAName;
            break;
        case PRODUCT_TYPE_B:
            productName = typeBName;
            break;
        case PRODUCT_TYPE_C:
            productName = typeCName;
            break;
        default:
            printf("Cannot find product type\n");
            free(product);
            return &nullProduct;
            break;
    }

    /* Copy product name */
    if ((product->name = (char *) malloc(strlen(productName) + 1)) == NULL) {
        printf("Cannot copy product name!!!\n");

```

```

        free(product);
        return &nullProduct;
    }
    strncpy(product->name, productName, strlen(productName) + 1);
    product->makeProduct = nullMakeProduct;
    product->do_something = productDoSomething;
    return product;
}

```

제품의 각종 정보를 채워주기 위해서 코드가 조금 길어졌지만 불필요하다고 생각되는 부분은 생략해도 된다. 원하는 제품의 타입이 없어도 오류를 돌려주지 않기 위해서 "Null Product"도 사용하고 있다. 나중에 제품을 사용하는 측에서 복귀값에 대해 "NULL"값 처리를 해주지 않아도 된다.

```

static void factoryDoSomething(FACTORY *factory, MESSAGE *msg) {
    printf("The factory name : %s, Message : %s\n", factory->name, msg->message);
    return;
}

```

```
FACTORY factory = { "Factory 01", makeProduct, factoryDoSomething };
```

팩토리 자체가 제품 생산 이외에 해야할 일은 "factoryDoSomething()"이 담당한다. 여기서도 필요하면 추가적인 코드를 넣어도 된다. 지금은 단순히 어떤 팩토리인지만 표시하도록 구현했다.

```

int main(void) {
    puts("Factory Method Pattern Example 01");
    MESSAGE msg = { "Product Factory Do Something Done!!!" };

    PRODUCT *productA = factory.makeProduct(PRODUCT_TYPE_A);
    PRODUCT *productB = factory.makeProduct(PRODUCT_TYPE_B);
    PRODUCT *productC = factory.makeProduct(PRODUCT_TYPE_C);
    PRODUCT *productNULL = factory.makeProduct(PRODUCT_TYPE_NONE); /* Cannot find product type */

    productA->do_something(productA, &msg);
    productB->do_something(productB, &msg);
    productC->do_something(productC, &msg);
    productNULL->do_something(productNULL, &msg);

    factory.do_something(&factory, &msg);

    return EXIT_SUCCESS;
}

```

앞의 코드는 하나의 팩토리에서 제품의 타입에 맞게 다양하게 제품을 생산하는 것을 보여준다. 각각의 제품은 자신의 역할을 수행하기 위해서 "do_something()"에 정해진 일을 처리하고 있다. 제품을 사용하는 측에서는 제품에서 제공하는 인터페이스만 알고 제품의 생산은 팩토리가 책임진다. 따라서, 제품을 어떻게 만들지와 제품의 내부 구조에 대해서는 사용하는 측에서는 알 수 없다.

어떤 제품을 만들 것인지 결정할 수 있다면, 만들어진 제품이 제공하는 동일한 인터페이스를 호출할 수 있다. 앞의 예에서 "PRODUCT" 타입의 포인터를 이용해서, 만들어진 제품에 대해서 동일한 인터페이스를 호출할 수 있다("do_something()"). 따라서, 제품과 제품을 사용하는 측에서는 정해진 인터페이

스를 통해서 요청을 주고 받을 뿐, 제품의 내부구조의 생성이나 변경에 대해서는 서로 독립성을 유지한다.

[결과]

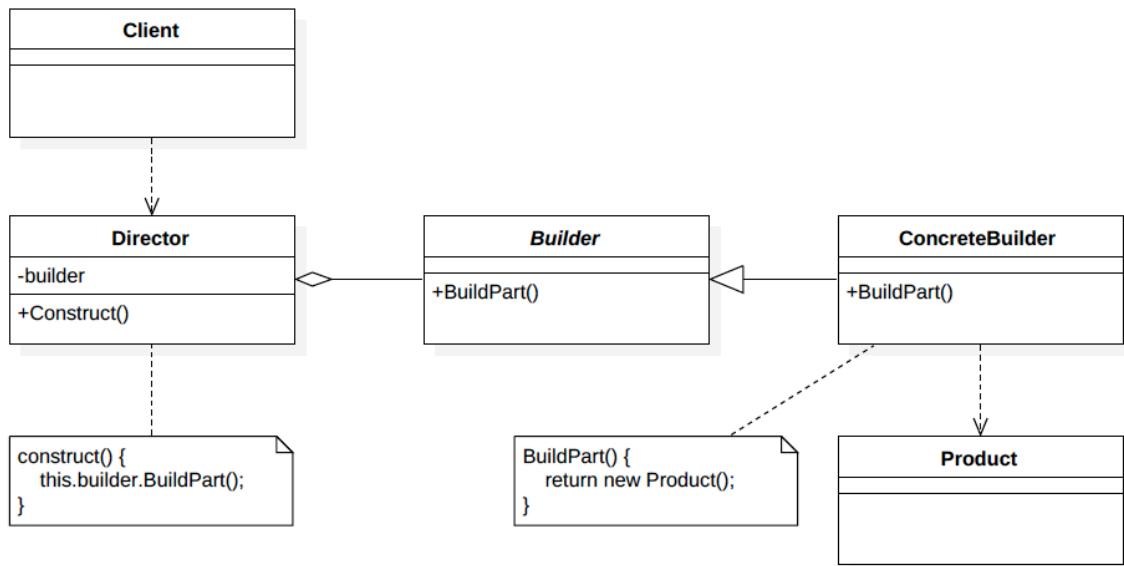
Factory Method Pattern Example 01

Cannot find product type

```
The product name : Product A, Message : Product Factory Do Something Done!!!
The product name : Product B, Message : Product Factory Do Something Done!!!
The product name : Product C, Message : Product Factory Do Something Done!!!
The product name : Null Product, Message : Product Factory Do Something Done!!!
The factory name : Factory 01, Message : Product Factory Do Something Done!!!
```

[빌더 패턴(Builder Pattern)을 이용한 개별 자료구조의 생성]

빌더는 여러 개로 이루어진 부품을 이용해서 하나의 제품을 만드는 개념을 사용한다. 즉, 각각의 부품을 이용해서 제품을 어떻게 만드는 것은 모르며, 그것을 만들어줄 수 있는 방법(인터페이스)만 알고 있다. 만들어 주는 측에 만들어야 하는 제품의 특징을 알려주고, 제품을 만드는 역할을 맡기는 방법을 사용한다.



클라이언트는 작업 담당자("Director")에게 제품을 만들도록 요청을 보낸다. 작업 담당자는 요청을 수행하기 위해서 각각의 부품을 담당하는 빌더("Builder")를 이용해서 부품의 생산을 의뢰한다 ("BuildPart()"). 실제 부품의 생산은 "ConcreteBuilder"가 담당하며, 만들어진 제품("Product")은 최종적으로 작업 담당자가 돌려받게 된다("this.builder.BuildPart()").

예제는 다양한 타입의 제품을 정의하고, 이것을 대신 생성해줄 빌더를 지정해서, 그것에 생산을 의뢰하는 과정을 보여준다. 필요하다면, 제품의 생산에 필요한 과정을 더 세분화 시켜서 나눌 수 있을 것이다. 물론, 제품을 직접 생산하는 것은 빌더의 역할이기에 만들어진 제품을 사용하는 것과 제품을 만드는 것은 분리될 수 있다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct message {
    char* message;
```

```

} MESSAGE;

typedef enum {
    PRODUCT_TYPE_A = 0,
    PRODUCT_TYPE_B,
    PRODUCT_TYPE_C,
    PRODUCT_TYPE_NONE
} PRODUCT_TYPE;

typedef struct product PRODUCT;

struct product {
    char *name;
    PRODUCT_TYPE type;
    void (*do_something)(PRODUCT *product, MESSAGE *msg);
};

void do_something(PRODUCT *product, MESSAGE *msg) {
    printf("The Product Name : %s, Message : %s\n", product->name, msg->message);
    return;
}

```

제품의 자료구조는 제품을 가지고 처리할 수 있는 일이 무엇인지 나타내기 위해서 "do_something()" 함수를 사용했다. 필요한 파라미터를 전달하기 위해서 "MESSAGE" 구조를 활용했지만, 추가적인 정보가 필요하다면 더 추가할 수 있을 것이다.

```

typedef struct builder BUILDER;

struct builder {
    char *name;
    PRODUCT_TYPE type;
    PRODUCT *(*makeProduct)(BUILDER *builder);
};

PRODUCT *builderMakeProduct(BUILDER *builder) {
    PRODUCT *product;

    if ((product = (PRODUCT *) malloc(sizeof(PRODUCT))) == NULL) {
        printf("Cannot make a product!!!\n");
        return NULL;
    }

    if ((product->name = (char *) malloc(strlen(builder->name) + 1)) == NULL) {
        printf("Cannot make a product name!!!\n");
        free(product);
        return NULL;
    }

    strncpy(product->name, builder->name, strlen(builder->name) + 1);
    product->type = builder->type;
    product->do_something = do_something;
}

```

```

    return product;
}

```

여기서는 특정 빌더는 특정 제품을 생산하는 것을 위탁받는다는 가정을 했다. 따라서, 자신이 위탁받아야 하는 제품의 타입을 알아야 하며("PRODUCT_TYPE"), 그 제품을 어떻게 만드는지 스스로가 결정해 주어야 한다. 외부에서는 그 제품을 어떻게 만드는지에 대해서 관심이 없다. 그리고, 관심을 가져서도 안될 것이다.

```

typedef struct maker MAKER;
struct maker {
    BUILDER *builder;
    void (*setBuilder)(MAKER *maker, BUILDER *builder);
    PRODUCT *(*makeProduct)(MAKER *maker);
};

void setBuilder(MAKER *maker, BUILDER *builder) {
    maker->builder = builder;
    return;
}

PRODUCT *makeProduct(MAKER *maker) {
    return maker->builder->makeProduct(maker->builder);
}

```

제품의 생산자("MAKER")는 제품의 빌더를 설정하고("setBuilder()"), 빌더에 대해서 제품의 생산을 의뢰하는 역할을 수행한다. 만들어진 제품을 필요로 하는 사용자 측에서는 생산자만을 통해서 제품의 생산을 의뢰하게 될 것이다.

```

int main(void) {
    puts("Builder Pattern Example 01"); /* prints Builder Pattern Example 01 */
    MESSAGE msg = { "Hello, Product!!!" };
    PRODUCT *product = NULL;
    MAKER maker = { NULL, setBuilder, makeProduct };
    BUILDER builderA = { "Product A", PRODUCT_TYPE_A, builderMakeProduct };
    BUILDER builderB = { "Product B", PRODUCT_TYPE_B, builderMakeProduct };
    BUILDER builderC = { "Product C", PRODUCT_TYPE_C, builderMakeProduct };

    maker.setBuilder(&maker, &builderA);
    product = maker.makeProduct(&maker);
    product->do_something(product, &msg);

    maker.setBuilder(&maker, &builderB);
    product = maker.makeProduct(&maker);
    product->do_something(product, &msg);

    maker.setBuilder(&maker, &builderC);
    product = maker.makeProduct(&maker);
    product->do_something(product, &msg);

    return EXIT_SUCCESS;
}

```

예제는 3종류의 제품에 대한 빌더를 각각 설정하고, 메이커가 제품의 타입에 따라 다른 제품을 생산하는 빌더를 설정하도록 만들었다.

빌더 패턴을 이용하는 것은 만드는 과정이 복잡한 특정 자료구조를 직접 만들어주는 대신, 해당하는 빌더를 이용해서 요청하는 방식으로 구현하기 위한 것이다. 이렇게해서 얻는 효과는 제품을 만드는 구체적인 부분을 숨겨 추상화 시켰다는 점과 빌더를 다른 것으로 교체 하더라도 동일한 인터페이스를 통해서 원하는 제품을 만들어서 사용할 수 있다는 것이다. 추가적인 제품을 만들어야 하는 경우, 해당하는 빌더를 구축해서 동일한 인터페이스를 이용해서 제품을 사용할 수 있다.

[결과]

Builder Pattern Example 01

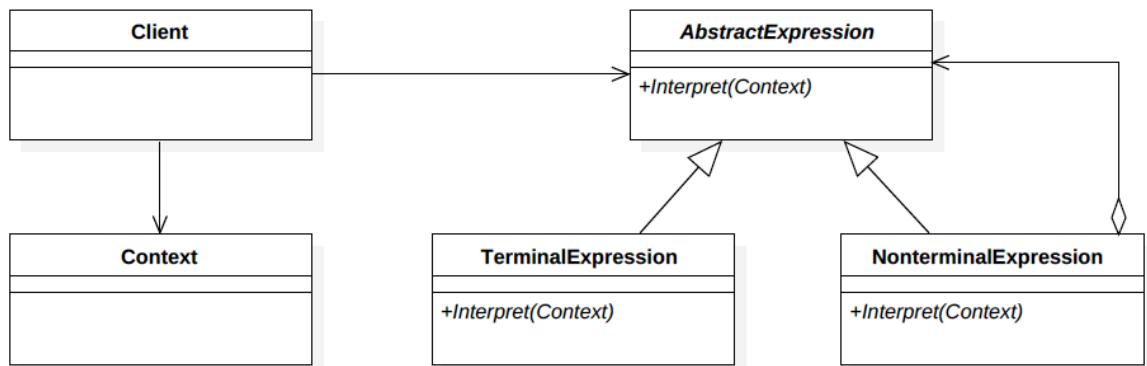
The Product Name : Product A, Message : Hello, Product!!!

The Product Name : Product B, Message : Hello, Product!!!

The Product Name : Product C, Message : Hello, Product!!!

[통역자 패턴(Interpreter Pattern)을 이용한 문장의 해석]

인터프리터 패턴은 전달받은 내용을 정해진 규칙에 맞게 번역하기 위해서 사용한다. 예를 들어, 입력된 수식을 나타내는 문자열을 해석해서 계산 결과를 만드는 것과 같은 경우에 사용할 수 있다. 또한, 특정 형식으로 만들어진 차트의 내용을 정리해서 결과 리포트를 생성하는데도 사용할 수 있을 것이다.



클라이언트는 문맥("Context")를 알고 있으며, 표현("AbstractExpression")을 해석하기 위해서 파라미터로 전달하게 된다. 표현은 다시 세부 표현의 형태로 나누어질 수 있으며, 더 이상 나누어 질 수 없는 표현("TerminalExpression")과 다른 표현들로 더 나누어질 수 있는 표현("NonterminalExpression")이 있을 수 있다. 각각의 표현은 자신을 해석하기 위한 자체 함수("Interpret()")를 가지고 넘겨받은 문맥을 해석하게 된다. 최종적으로 해석된 결과는 클라이언트로 전달될 것이다.

예제는 수식을 표현하는 문자열을 입력받아 해석한 후에 결과값을 계산하는 것을 보여줄 것이다. 예제는 미리 수식을 입력받아 원하는 형태의 입력으로 이미 변환 했다고 가정했다. 예를 들어, "10 - (20 + 30)"이라는 수식이 있다면, "10, 20, 30, +, -"와 같이 좌측에서 우측으로 정리되어 있다고 가정한다(일반적으로 괄호를 사용한 사칙 연산과 같은 구문의 해석에서 흔히 사용 되는 형태).

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <stdbool.h>
```

```
#define MAX_STACK_SIZE 100
typedef struct stack STACK;

struct stack {
    char *inputs[MAX_STACK_SIZE];
    int top;
    void (*push)(STACK *stack, char *input);
    char *(*pop)(STACK *stack);
};

static void push(STACK *stack, char *input) {
    if (stack->top >= MAX_STACK_SIZE) {
        printf("Cannot push into the stack!!!\n");
        return;
    }
    stack->inputs[stack->top] = input;
    stack->top++;
    printf("Push : %s\n", input);
    return;
}

static char *pop(STACK *stack) {
    if (stack->top <= 0) {
        printf("Cannot pop the empty stack!!!\n");
        return NULL;
    }
    stack->top--;
    printf("Pop : %s\n", stack->inputs[stack->top]);
    return stack->inputs[stack->top];
}

STACK stack = { { }, 0, push, pop };
```

주어진 수식은 연산자를 중심으로 깊이 우선 검색에 의해서 만들어진 것이다. 즉, 좌에서 우측으로 읽으면서 연산자를 만날 때, 이미 두 개의 좌우 피연산자 값을 읽은 상태가 된다. 따라서, 연산자를 만났을 때 이미 읽은 값을 역으로 빼내기 위해서 스택과 같은 자료구조를 이용했다.

예를 들어, "10->20->30"이라는 순으로 문자열의 입력을 받게되면, 다음에 "+"와 같은 연산자가 올 경우, 이미 저장된 값 중에서 가장 최근에 저장된 두개의 값인 "20"과 "30"을 꺼내게 된다. 이를 연산자의 의미에 따라 계산하게 된다. 계산된 결과는 다시 스택에 저장되어 다른 연산자를 만날 때까지 대기하게 된다.

```
typedef struct operand OPERAND, OPERATOR;

struct operand {
    int (*interpret)(char *input);
};

static int operandInterpret(char *input);
static int operatorInterpret(char *input);
```

```
OPERAND operand = { operandInterpret };
OPERATOR operator = { operatorInterpret };
```

사실, 계산해야 할 값들은 트리(Tree)의 가장 말단(Terminal)에 속하며, 그것을 연결하는 것이 연산자를 값으로 가지는 노드들이다. 하지만, 값을 계산하는 측면에서는 동일한 인터페이스("Interpret")로 가능하기에 같은 값으로("operand", "operator") 선언해 주었다. 인터페이스는 동일하지만 구현은 각각이 어떻게 해석 되는가에 따라 다르다.

```
static int operandInterpret(char *input) {
    /* 정수 값으로 변경한다. */
    return atoi(input);
}

static int operatorInterpret(char *input) {
    int left = 0;
    int right = 0;

    /* 덧셈인지 확인한다. */
    if (strncmp(input, "+", strlen(input)) == 0) {
        right = operand.interpret(stack.pop(&stack));
        left = operand.interpret(stack.pop(&stack));
        return left + right;
    }

    /* 뺄셈인지 확인한다. */
    if (strncmp(input, "-", strlen(input)) == 0) {
        right = operand.interpret(stack.pop(&stack));
        left = operand.interpret(stack.pop(&stack));
        return left - right;
    }

    /* 잘못된 연산자를 처리한다. */
    printf("Cannot parse the input : %s\n", input);
    exit(1);
}
```

연산자인 경우에는 스택에 마지막에 저장된 문자열을 불러와서 값으로 전환하고(interpret), 계산된 결과 값을 돌려주도록 했다. 만약, 값을 의미하는 문자열이라면, 단순히 문자열을 숫자로 변경해서 돌려주도록 만들었다.

```
typedef struct expression EXPRESSION;
struct expression {
    int (*interpret)(char *input[]);
};

static bool isOperator(char *input) {
    if (strncmp(input, "+", strlen(input)) == 0)
        return true;
    }
    if (strncmp(input, "-", strlen(input)) == 0)
        return true;
}
```

```

    }
    return false;
}

#define MAX_INTEGER_SIZE 15
static char *inttoascii(int input) { /* 정수값을 아스키 값으로 변환한다. */
    char *str;

    if ((str = (char*) malloc( MAX_INTEGER_SIZE)) == NULL) {
        printf("Cannot allocate memory for storing values\n");
        return NULL;
    }
    sprintf(str, "%d", input);
    return str;
}

```

앞의 코드는 연산자인지를 결정하는 간단한 함수와 숫자를 문자열로 변경하는 함수를 선언한 것이다. 우리가 지금 처리하고 있는 데이터의 형식이 문자열이기 때문에, 숫자로 변환해서 계산한 후에 스택에 다시 넣기 위해서는 문자열로 변환해 주어야 한다. "itoa()"라는 비 표준 함수가 이런 역할을 할 수 있지만, 여기서는 새로 구현하는 방식으로 처리했다("inttoascii()"). 최대 음수를 표현하기 위해서 "-"를 포함해서 15자의 문자를 기록할 수 있도록 만들었다. 문자열 변환은 "sprintf()"를 이용했다.

```

static int expressionInterpret(char *input[]) {
    int i = 0;
    int result = 0;

    while (input[i] != NULL) {
        if (isOperator(input[i])) {
            printf("Operator : %s\n", input[i]);
            result = operator.interpret(input[i]);
            stack.push(&stack, inttoascii(result));
            i++;
            continue;
        }
        stack.push(&stack, input[i]);
        i++;
    }
    result = atoi(stack.pop(&stack));
    return result;
}

```

EXPRESSION expression = { expressionInterpret };

하나의 수식은 연산자와 두 개의 값으로 이루어진다. 그것을 처리하기 위해서 만들어진 것이 "expression"이다. 따라서, 입력으로 받아들이는 값은 계산식을 표현하는 문자열 전체가 된다. 더 이상 처리할 자료가 없을 때까지 반복적으로 하나씩 문자열을 읽어서 처리하도록 만들었다. 계산된 결과는 다시 스택에 저장되며 최종 결과는 마지막으로 스택에 남은 값을 돌려주는 것으로 처리했다.

```

typedef struct interpreter INTERPRETER;
struct interpreter {
    EXPRESSION *expr;

```

```

int (*interpret)(INTERPRETER *interpreter, char *input[]);
};

static int interpreterInterpret(INTERPRETER *interpreter, char *input[]) {
    return interpreter->expr->interpret(input);
}

```

인터프리터는 자신이 알고 있는 표현식("Expression")을 처리해주는 역할을 한다. 즉, 클라이언트의 요청을 받아서 정해진 규칙을 이용해서 처리한 후, 결과값을 클라이언트에 돌려주게 된다. 입력은 수식 형태 이기에, "Expression"을 호출해서 번역하도록 요청한다.

```

INTERPRETER interpreter = { &expression, interpreterInterpret };

int main(void) {
    puts("Interpreter Pattern Example 01"); /* prints Interpreter Pattern Example 01 */
    char *input1[] = { "10", "20", "30", "+", "-", NULL }; /* = 10 - ( 20 + 30 ) = -40 */
    char *input2[] = { "100", "200", "300", "+", "400", "500", "-", "+", "-", NULL }; /* 100 - (( 200 + 300 ) + ( 400 - 500 )) = -300 */

    printf("The result value : %d\n", interpreter.interpret(&interpreter, input1));
    printf("=====\\n");
    printf("The result value : %d\\n", interpreter.interpret(&interpreter, input2));

    return EXIT_SUCCESS;
}

```

예제는 두 개의 수식 입력에 대해서 어떻게 결과가 계산 되는지를 보여준다. 위의 예에서와 같이 마치 문법과 같은 규칙을 만들어, 주어지는 데이터를 해석한 후에 올바른 결과값을 돌려준다는 것을 알 수 있을 것이다. 하지만, 위의 프로그램은 몇 가지 단점을 가지고 있으며 아래와 같다.

[개선할 부분]

1. 중간 결과를 저장하기 위해서 할당한 메모리를 해제하지 않는다.
2. 스택 크기가 한정되어 있어서 너무 복잡한 식은 계산하지 못한다.
3. 아직 사칙연산이 전부다 구현된 것은 아니다.(“, /“).
4. 숫자를 표현하는 문자 길이는 “-”를 포함해서 15자에 한정된다.
5. 입력받은 문자열이 정렬되어 있다고 가정했다.
6. 스택의 오버플로우(Overflow)나 언더플로우(Underflow) 오류를 처리해 주어야 한다.
7. "inttoascii()" 함수는 컴파일러에 따라 "itoa()"로 변경가능하다.

[결과]

Interpreter Pattern Example 01

```

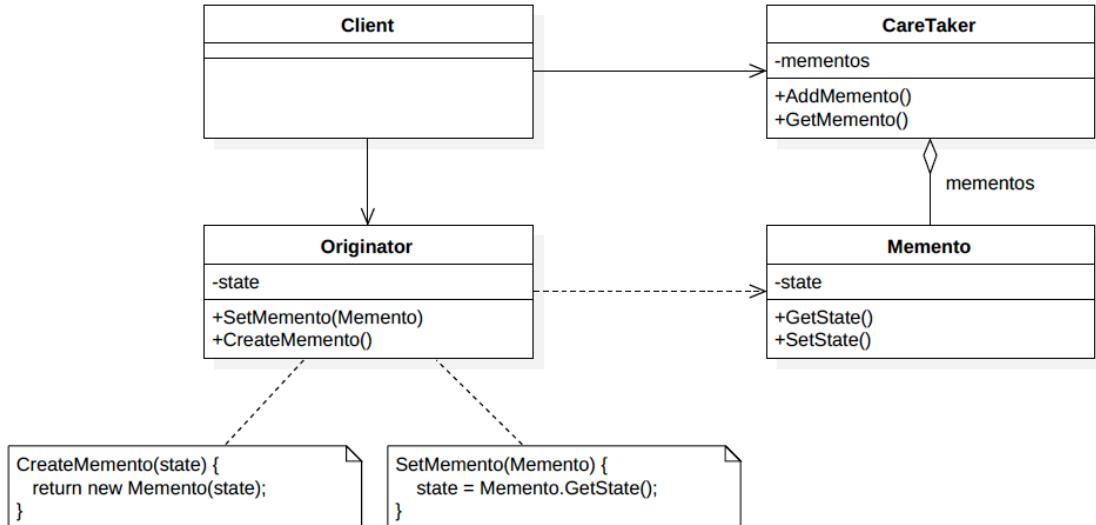
Push : 10
Push : 20
Push : 30
Operator : +
Pop : 30
Pop : 20
Push : 50
Operator : -
Pop : 50
Pop : 10

```

```
Push : -40
Pop : -40
The result value : -40
=====
Push : 100
Push : 200
Push : 300
Operator : +
Pop : 300
Pop : 200
Push : 500
Push : 400
Push : 500
Operator : -
Pop : 500
Pop : 400
Push : -100
Operator : +
Pop : -100
Pop : 500
Push : 400
Operator : -
Pop : 400
Pop : 100
Push : -300
Pop : -300
The result value : -300
```

[메멘토 패턴(Memento Pattern)을 이용한 “Undo”의 구현]

메멘토 패턴은 상태를 저장하기 위해서 사용한다. 즉, 현재 상태를 저장해서 나중에 그 상태로 다시 복구하기 위해서 사용된다. 예를 들어, 게임의 상태를 저장하거나, 혹은 문서 작성시 이전의 상태로 돌아가는 경우에 사용할 수 있다. 메멘토가 의미하는 바와 같이 "과거의 기억을 되살릴 만한 것"을 미리 저장해서 유지하다가, 필요한 순간 그것을 다시 가져와서 사용할 수 있다. 하지만, 저장되는 정보는 메멘토와 메멘토에 저장을 요청하는 쪽만 알고 있으며, 그것을 저장하고 관리하는 쪽에서는 어떤 정보가 저장되는지는 몰라야 한다.



클라이언트는 새로운 메멘토를 생성하기 위해서 “Originator”를 알고 있다. “Originator”의 역할은 자신의 현재 상태(“state”)를 이용해서 새로운 메멘토(“Memento”)를 만들거나(“CreatorMemento()”), 전달받은 메멘토를 이용해서 저장된 상태로 복구하는(“SetMemento()”) 역할을 수행한다. “Originator”에서 만들어진 메멘토는 “CareTaker”가 관리(“mementos”)하며, 메멘토의 추가(“AddMemento()”) 및 얻기(“GetMemento()”)가 가능하다. 따라서, 메멘토는 상태를 저장하고 복구하는데 효과적으로 활용할 수 있는 방법을 제공한다고 생각할 수 있다.

```

#include <stdio.h>
#include <stdlib.h>

typedef enum {
    STATE_A = 0,
    STATE_B,
    STATE_C,
    STATE_NONE,
} STATE;

typedef struct memento MEMENTO;
struct memento {
    STATE state;
    STATE (*getState)(MEMENTO *memento);
};

static STATE getState(MEMENTO *memento) {
    return memento->state;
}

static STATE nullGetState(MEMENTO *memento) {
    return STATE_NONE;
}

MEMENTO nullMemento = { STATE_NONE, nullGetState };
  
```

여기서는 저장되는 정보를 단순히 상태("state")로 표현했다. 그리고, 메멘토 자체는 자신의 정보를 얻을 수 있는 함수만 제공한다("getState()"). 저장되는 메멘토들은 스택과 같은 것을 이용해서 관리되기 때문에, 스택에서 더 이상 되돌릴 메멘토가 없다면, "NULL"을 표현하는 자료구조를 이용하도록 만들었다 ("nullMemento");

```

typedef struct originator ORIGINATOR;

struct originator {
    STATE state;
    void (*print_state)(ORIGINATOR *originator);
    void (*setState)(ORIGINATOR *originator, STATE state);
    MEMENTO *(*createMemento)(ORIGINATOR *originator);
    void (*setMemento)(ORIGINATOR *originator, MEMENTO *memento);
};

void print_state(ORIGINATOR *originator) {
    printf("The Originator is : ");
    switch (originator->state) {
        case STATE_A:
            printf("in A State\n");
            break;
        case STATE_B:
            printf("in B State\n");
            break;
        case STATE_C:
            printf("in C State\n");
            break;
        default:
            printf("in Unknown State\n");
            break;
    }
}

void originatorSetState(ORIGINATOR *originator, STATE state) {
    originator->state = state;
    return;
}

static MEMENTO *createMemento(ORIGINATOR *originator) {
    MEMENTO *memento;

    if ((memento = (MEMENTO *) malloc(sizeof(MEMENTO))) == NULL) {
        printf("Cannot allocate memento!!!\n");
        return &nullMemento;
    }
    memento->state = originator->state;
    memento->getState = getState;
    return memento;
}

static void setMemento(ORIGINATOR *originator, MEMENTO *memento) {

```

```

if (memento != &nullMemento) {
    originator->state = memento->getState(memento);
}
/* Do nothing for null memento */
return;
}

```

"Originator"는 상태를 가지며, 자신의 상태를 설정하기 위한 함수("setState()"), 현재 상태를 출력하는 함수("print_state()"), 새로운 메멘토를 생성해서 자신의 상태를 저장하는 함수("createMemento()"), 메멘토에서 상태를 복구하는 함수("setMemento()")를 제공한다. 이들 각각의 인터페이스 중에서 상태를 설정하는 것과 상태를 출력하는 것은 반드시 있어야 할 필요는 없지만 코딩의 편의 때문에 추가했다.

```
typedef struct careTaker CARETAKER;
```

```
#define MAX_SAVED_MEMENTO 100
struct careTaker {
    int top;
    MEMENTO *savedMemento[MAX_SAVED_MEMENTO];
    void (*addMemento)(CARETAKER *careTaker, MEMENTO *memento);
    MEMENTO *(*getMemento)(CARETAKER *careTaker);
};
```

```
static void addMemento(CARETAKER *careTaker, MEMENTO *memento) {
    if (careTaker->top >= MAX_SAVED_MEMENTO) {
        printf("Cannot add memento!!!\n");
        return;
    }
    careTaker->savedMemento[careTaker->top] = memento;
    careTaker->top++;
    return;
}
```

```
static MEMENTO *getMemento(CARETAKER *careTaker) {
    if (careTaker->top <= 0) {
        printf("Empty memento!!!\n");
        return &nullMemento;
    }
    careTaker->top--;
    return careTaker->savedMemento[careTaker->top];
}
```

```
ORIGINATOR originator = { STATE_NONE, print_state, originatorSetState,
                           createMemento, setMemento };
CARETAKER careTaker = { 0, { }, addMemento, getMemento };
```

"CareTaker"의 역할은 전달받는 메멘토를 관리하는 것이다. 즉, 메멘토를 저장하는 함수("addMemento()"), 저장된 메멘토를 가장 최근에 저장된 것부터 가져오는 함수("getMemento()")등을 가진다. 스택을 이용해서 가장 최근에 저장된 것이 가장 먼저 나오도록 구현했다. 메멘토들은 동적으로 생성되도록 만들었지만, "Originator" 와 "CareTaker"는 구현 편의상 정적으로 선언해서 사용했다.

```
int main(void) {
```

```

puts("Memento Pattern Example 01");
MEMENTO *memento;

originator.print_state(&originator);
originator.setState(&originator, STATE_A);
originator.print_state(&originator);
memento = originator.createMemento(&originator);
careTaker.addMemento(&careTaker, memento);

originator.setState(&originator, STATE_B);
originator.print_state(&originator);
memento = originator.createMemento(&originator);
careTaker.addMemento(&careTaker, memento);

originator.setState(&originator, STATE_C);
originator.print_state(&originator);
memento = originator.createMemento(&originator);
careTaker.addMemento(&careTaker, memento);

printf("===== Undo =====\n");
originator.setMemento(&originator, careTaker.getMemento(&careTaker));
originator.print_state(&originator);
originator.setMemento(&originator, careTaker.getMemento(&careTaker));
originator.print_state(&originator);
originator.setMemento(&originator, careTaker.getMemento(&careTaker));
originator.print_state(&originator);
originator.setMemento(&originator, careTaker.getMemento(&careTaker)); /* Empty
memento!!! */
originator.print_state(&originator);

return EXIT_SUCCESS;
}

```

예제는 상태를 3개 정의할 수 있도록 했으며, 각각을 저장 했다가 다시 복구하는 과정을 보여주게 만들었다. 필요하다면 저장할 상태의 필드를 추가하거나 상태를 더 추가해도 상관없다. 초기에는 상태를 정해주지 않았기에, 설정할 수 있는 함수를 제공했다("setState()"). 결과에서 보듯이 저장된 메멘토들은 역순으로 하나씩 끄집어내서 상태를 변경해주고 있다. 더 이상 복구할 메멘토가 없을 때는 비었다는 것을 표시하고, "Null" 메멘토를 이용하도록 만들었다.

[결과]

```

Memento Pattern Example 01
The Originator is : in Unknown State
The Originator is : in A State
The Originator is : in B State
The Originator is : in C State
===== Undo =====
The Originator is : in C State
The Originator is : in B State
The Originator is : in A State
Empty memento!!!

```

The Originator is : in A State

지금까지 이곳에서 보여준 디자인 패턴 및 예제 들은 단지 C언어에서 활용하기 위한 한가지 방법일 뿐이다. 따라서, 디자인 패턴을 이해하고 그것을 적절하게 활용하는 것은 해결하려는 문제에 따라 다양한 방법이 있을 수 있다. 단지, 반복되는 문제의 유형에 따른 해결책을 제시하고자 예를 만들었을 뿐이다. 따라서, 이곳에서 보여준 예제들을 활용하거나 다른 예제를 찾을 수 있으면, 자신이 해결해야 하는 문제에 대한 익숙한 해결책을 좀 더 빨리 찾아 낼 수 있을 것이다.

중요한 점은 디자인 패턴 역시 코드에서 발생하는 중복을 제거하고, 추가 및 확장을 위한 구조를 제공하기 위한 해결책 이라는 점이다. 이미 이야기 했듯이 중복은 버그의 온상이며, 추가 및 확장은 소프트웨어가 존재하는 한 변하지 않는 특성이다.

7. 최적화

프로그래밍이란 코딩 언어를 통해서 대화를 나누는 것이다. 대화의 상대는 프로그램을 실행시킬 컴퓨터가 아니라, 같이 일하고 있는 동료 개발자나 자기 자신이 될 수 있다. 언어는 대화의 수단이며, C언어도 예외가 아니다. "읽기 쉬운 코드가 좋은 코드"라는 말은 사용하는 언어를 불문하고 항상 진리다.

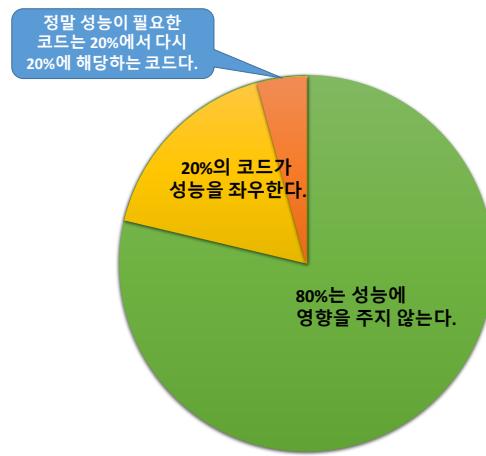
"구조화된 코드는 최적화가 가능하지만, 최적화된 코드는 구조화가 불가능하다"라는 말이 있다. 즉, 구조화된 코드는 읽기도 쉽고 이해하기 쉬워서 최적화를 손쉽게 할 수 있지만, 최적화 되었다고 말하는 코드들은 대부분 구조적이지 못하다(모듈화나 계층화가 안되어 있다.). 따라서, 최적화된 코드(?)를 구조화시키기 위해서 필요한 코드 수정은 새로운 버그들을 쉽게 만들어 내고 만다.

읽기 어려운 코드를 최적화 시키려고 노력하기보다는, 오히려 읽기 쉬운 코드를 먼저 작성하려고 시간을 투자하는 것이 최적화에도 도움이 된다. 일반적으로 최적화는 구조화와 반대되는 성격을 가지고 있지만, 최적화를 돋기 위해서는 먼저 이해하기 쉬운 코드 부터 작성하는 것이 지름길인 것이다. 최적화를 하기 전에 먼저 아래와 같은 고민해야 할 것이다.

- 01. 최적화를 위해서 코딩 하지 말라.**
- 02. 정말 최적화가 필요한지 검토하라.**
- 03. 최적화하기에 앞서 측정 하라.**
- 04. 측정 결과에서 문제가 되는 부분만 최적화 하라.**

최적화하기 위해서 코딩 했다고 하는 대부분의 코드는 가독성이 떨어지거나(코딩한 사람만 알아보거나), 실제로 최적화 되지 않는 경우가 많다. 물론, 코딩 룰에서 최적화를 위한 방법으로 정한다면 이야기는 조금 달라지지만, 최적화를 위해서 코딩하는 것은 유지보수나 수정 비용만 높일 뿐이다. 최적화 때문에 계층화나 추상화와 같은 개념이 생략되고 의존성이 높아지는 코드를 만들게 되면, 정말 최적화나 구조변경이 필요한 순간에는 도움이 되지 못할 뿐더러 심한 경우에는 오히려 최적화의 걸림돌로 작용하는 경우가 많다.

먼저 해야 할 일은 정말 최적화가 필요한지 확인하는 것이다. 충분히 코드가 빠르게 실행되고 있고, 공간도 적게 차지한다면, 굳이 최적화를 할 필요가 없을 수도 있다. 임베디드 시스템과 같은 경우 실제로 최적화를 해야 한다고 이야기하지만, CPU의 노는 시간이(Idle Time) 70% 이상을 차지하는 경우도 종종 있다. 이것은 대부분 I/O에 관련된 대기시간 때문에 발생하는 것이지, 코드를 최적화해서 수행속도를 높여야 할 이유는 되지 않는다. 극한의 최적화가 필요한 순간도 있지만, 사실 그 정도쯤 되면 다음번 제품에서 다른 더 성능이 높은 CPU를 사용하거나, RAM을 늘려야 할지도 모른다. 최적화로 인해서 발생하는 복잡도 증가나 가독성이 낮아지는 코드는 차기 제품까지 영향을 주지 않도록 항상 조심해야 할 것이다.



최적화를 하기 위해서는 먼저 어느 곳을 최적화 해야 할지 정해야 한다. 예를 들어, 전체 수행 시간동안 100번 수행되는 A함수와 10,000번 사용되는 B함수가 있다고 가정하자. 각각의 수행 시간은 10초와 5초라고 할 때, A함수는 전체 사용시간이 1,000초가 되고 B함수는 50,000초가 된다. 따라서, B함수를 최적화 하는 것이 오히려 A함수를 최적화 시키는 것 보다 훨씬 가치가 있을 것이다. 하지만, 여기서도 문제는 있다. 즉, 만약 A 함수가 사용자에게 민감한 일을 처리하고 있고 B함수는 그렇지 않다면, A함수를 최적화 할 필요도 있을 것이다.

따라서, 최적화는 세 가지를 다 고려해서 적절한 코드를 선택해야 한다는 것을 알 수 있다. 각각은 “사용자에게 주는 가치”, “빈도”, “수행 시간”이 될 수 있을 것이다. 만약, 다른 것들을 더 고려할 것이 있다면 추가해서 좋지만, 우리가 이용할 수 있는 도구(프로파일러[Profiler]와 같은)는 일반적으로 빈도와 수행 시간만 알려줄 수 있을 것이다. 나머지는 개발자의 판단에 맡겨야 할지도 모른다.

최적화 노력의 대부분이 최적화가 필요없는 부분에 투자되고 있는 이유가 바로 여기에 있다. 즉, 측정 없이 최적화를 하려고 노력하기 때문이다. 오히려 측정을 먼저하게 되면 어디를 최적화 해야 할지 명확히 알 수 있으며, 그 곳에 노력을 집중하는 것이 ROI(Return of Investment)를 높이는 길이다. “8:2의 법칙”이 이 때도 유용한 것은 “최적화 노력의 대부분은 최적화가 필요없는 부분에 투자 된다”는 말처럼 보이기 때문이다. 그리고, 다시 최적화 할 필요가 있는 코드의 20%만이 실제 성능 향상에 크게 기여할 수 있을 것이라는 것도 추측할 수 있을 것이다.

컴파일러의 최적화 옵션을 프로젝트의 후반에 높이는 것은 위험하다. 갑자기 잘 동작 하던 코드들이 제대로 동작하지 않을 수도 있다. 따라서, 컴파일러의 최적화 옵션은 가능한 과제의 초기부터 높이 설정하는 것이 좋다. 예를 들어, "GCC"와 같은 경우에는 "-O3" 옵션을 이용해서 "Release"모드로 빌드하고 테스트 할 필요가 있다. 만약, 특정 CPU를 사용하는 임베디드 시스템과 같은 경우에는 컴파일러도 선택적으로 사용할 수 있는데, 일반적으로 상업적으로 구매할 수 있는 컴파일러가 최적화(코드 크기 및 시간적인 부분에서)를 더 잘하는 경향이 있다. ARM프로세서 용 프로그램을 개발하는 할 때는 "GCC"보다는 "ARM"에서 제공하는 컴파일러가 더 높은 성능을 가져다 줄 가능성이 높다. 필요한 성능이 컴파일러의 변경으로 가능하다면, 굳이 다른 최적화를 하지 않아도 될 것이다.

컴파일러의 최적화는 크기(Size)와 시간(Time)의 설정에 따라 달라질 수 있다. 시간에 대한 최적화는 반복문의 처리와 같은 곳에서 효과를 발휘하며, 크기에 대한 최적화는 함수 호출의 처리에 영향을 주게 된다. 만약, 인라인(Inline)으로 선언된 함수가 있다면, 크기의 최적화에서 일반 함수의 호출로 대체되고, 시간에 대한 최적화에서는 인라인으로 코드 내에 삽입 될 것이다. 예를 들어, 반복문의 처리에서 풀어해치(Unrolling)는 최적화는 시간에 대한 최적화이다.

컴파일러의 최적화 옵션을 제대로 점검하지 않으면, 과제의 후반에 코드의 분량이 커져서 컴파일러의 최적화 옵션을 제대로 활용하지 못하는 경우도 발생한다. 이때는 최적화 옵션을 적용할 수 있는 부분들을

찾아서 각각을 별도의 모듈로 분리시켜서 컴파일하는 것도 고려해 볼 수 있다. 물론, 제대로 하는 것은 초기부터 최적화 옵션을 높이 설정하고 제대로 동작 하는지 검증하는 것임을 잊어선 안된다. 나중에 최적화 옵션을 높일 경우 종종 곤란한 경우를 당할 수 있다.

[최적화 전에 최적화가 필요한지 확인하라.]

예전에 실제 경험한 일이지만, 임베디드 시스템에서 사용 중인 데이터 베이스(Database)가 제대로 성능이 나지 않는다고, 새로운 데이터 베이스 엔진을 개발하는 과제를 만든 경우를 본적이 있다. 물론, 문제는 데이터 베이스 엔진에 있었던 것이 아니라, 그것을 사용하는 개발자가 잘못된 질의를 데이터 베이스에 보내고 있었다. 데이터 베이스에 대한 질의를 조금 바꾼 결과, 새롭게 데이터 베이스 엔진을 만들어야 할 필요가 거의 사라져 버리고 말았다. 무엇이 안된다고 할 때, 혹시 우리가 잘못 사용하고 있는 것은 아닌지 먼저 살펴야 하는 것이다.

이와 같은 경우는 흔히 있을 것이다. 특히, 다른 사람이 개발한 코드를 사용할 경우, 제대로 된 사용법을 모르고 사용하는 경우가 있다. 따라서, 최적화를 하기 위해서는 정말 최적화가 필요한지를 반드시 먼저 확인해야 한다. 혹시 잘못 사용하고 있는 것은 아닌지? 필요하지 않은데 지나치게 많은 일을 하고 있는 것은 아닌지 먼저 확인해야 한다.

앞에서 들었던 예와는 조금 다르지만, 인터럽트(Interrupt)와 폴링 방식을 잘못 사용해서 발생하는 CPU의 과도한 점유도 문제가 되었던 적이 있다. UART와 같은 장치의 특정 상태를 읽어서 데이터의 전송이 완료된 것을 파악해야 했었는데, “대기(Busy Waiting)”하는 방법을 사용해서 CPU를 완전히 차지한 경우다. 실시간 운영체제(RTOS)에서는 이런 상황에 다른 작업을 못하게 막아 전체적으로 시스템의 반응 속도를 늦게 만들거나, 다른 작업의 처리가 불가능하게 만들 수도 있다. 이 때는 인터럽트를 이용해서 전송의 완료를 전달받는 방식이 더 효과적이다.

물론, 최적화하는 과정은 쉽지 않다. 선입견을 가지고 접근하면 최적화 해야 할 부분을 찾아낼 수 없을지도 모른다. 모든 가정을 일단 하나씩 검증해 나갈 필요가 있으며, 올바르지 못한 가정을 하나씩 실험해서 지워 나가야 한다. 이것 역시 “느낌”이 아니라 실제로 측정한 데이터를 통해서 하나씩 순차적으로 해야한다. 갑작스런 큰 코드의 변화는 시스템을 불안정한 상태로 만들 가능성이 있으며, 수정된 내용이 정말 최적화와 관련이 있지 않을 수도 있기 때문이다.

[최적화를 위한 접근 방법]

최적화 해야 할 코드를 찾았다면, 이제는 해당하는 코드를 어떻게 최적화 할 것인지 알아야 한다. 이때도 너무 많은 수정은 최적화를 방해하기에, 조금씩 고치고 조금씩 측정해나가면서 진행해야 한다. 다음은 최적화를 위한 접근 방법들을 나열한 것이다.

01. 컴파일러를 믿어라.

; 컴파일러는 오래 동안 발전해 왔다. 이미 자주 발생할 수 있는 다양한 문제점을 해결해 왔으며, 특히 성능에 대해서는 최적화를 사람보다 잘한다. 컴파일러 자체도 다양하게 있기에 적절한 컴파일러를 사용하는 것도 중요하다. 경우에 따라 최적화를 더 잘하는 컴파일러를 사용해서 목표한 성능을 얻을 수 있다면 최적화에 들어가는 노력을 크게 줄일 수 있다.

02. 구조적으로 변경할 수 있는 방법이 없는지 찾는다.

; 필요없는 부분을 없애는 것이 최적화의 시작이다. “완벽이란 더 이상 뺄 것이 없는 상태”를 말하며, 최적화도 크게 다르지 않다. 더 일을 적게 하는 것이(혹은, 필요한 일만 하는 것이) 최적화하는 길이다. 코드를 적게 작성하는 것이 버그도 적게 발생 시킨다.

03. 관련이 없는 코드는 분리한다.

; 서로 같이 동작할 필요가 없는 코드는 같이 있을 필요가 없다. 생각보다 이런 코드들은 많이 있을 수 있으며, 나중에 미뤄서 실행하거나 한번만 실행해 주어도 된다. 혹은, 시간이 날 때 실행해도 상관없을 수 있다. 관련이 없는 코드를 최대한 줄이면 그 만큼의 시간과 공간을 절약할 수 있다.

04. 필요없는 코드를 제거한다.

; 필요없이 여러 번 할당하거나 쓰임새가 명확히 없는 변수들이 코드에 남아 있는 경우 있다. 변수는 하나의 역할과 목적으로만 사용되어야 하며, 사용되지 않는 변수들은 전부 삭제해야 한다. 가능한 필요없는 코드는 전부 제거하도록 한다. 필요한 코드를 최적화하는 일도 크기 때문에, 필요없는 코드는 가능하다 없애도록 한다. 물론, 필요 없다고 말할 수 있는 코드가 어떤 코드인지는 팀의 판단에 맡겨야 할 것이다. 적어도 “나중에 사용될 가능성이 있는 코드”는 일단 필요없는 코드로 보는 것이 좋다. 현재 사용되는 코드만 남겨둔다.

05. 될 수 있으면 대기(Busy waiting)를 줄이고, 인터럽트나 이벤트 방식으로 변경한다.

; 동기화가 반드시 필요한 일을 제외하면, CPU 사이클을 소모하면서 대기하는 것은 다른 방식으로 변경해야 한다. 실시간 시스템과 같은 경우에는 CPU의 사용률이 70%를 넘으면, 실시간 작업들이 데드라인(Dead Line)을 놓칠 가능성이 높다. 비동기 처리가 대부분 경우 더 높은 처리률을 달성하는데 도움이 된다. 동기화된 처리는 반드시 처리 결과를 확인해야 할 필요가 있는 경우에 사용해야 한다.

06. “floating point”계산을 줄이거나 없앤다.

; “Floating Point” 연산은 항상 정수 연산보다 오버헤드가 크다. 차라리 정수로 만들어서 계산하면 더 빨리 할 수 있다. 소수점이하 만큼의 10을 곱해서 정수로 만들면 계산 속도를 높일 수 있다. 운영체제나 디바이스 드라이버, 임베디드 시스템의 응용프로그램을 만든다면, “Floating Point”연산은 거의 사용하지 않는다. 따라서, 입력이 항상 일정하고 중간 계산 과정이 필요한 경우에는 테이블을 이용하는 것도 최적화를 위한 한가지 방법이 될 수 있다.

07. 조건문을 간략하게 만들거나 없앤다.

; 조건문은 가능한 줄여서 사용하는 것이 좋다. 미리 설정할 수 있다면 최대한 조건문을 제거하도록 한다. 조건문의 잣은 사용은 가독성에 영향을 주며, 테스트도 어렵게 만들 수 있기에 가능한 사용하지 않도록 한다. 파이프 라인을 가지고 있는 CPU에도 조건문은 좋지 않은 영향을 줄 수 있다. 조건문의 중첩을 줄이고 단순한 조건으로 바꾸는 것이 성능에도 도움이 된다.

08. 분기가 많다면 테이블을 이용해서 비교를 줄인다.

; 예를 들어, “switch()”문이 처리해야 할 “case”가 많다면, 함수 포인터를 가진 배열로 대체할 수 있다. 즉, “switch()”문의 입력이 0에서 시작하고 연속적인 값을 가지고 정수라면 배열을 이용해서 실행해야 할 함수들의 포인터를 바로 호출하는 방법을 사용할 수 있다. “case”하나가 늘어날수록 복잡도가 증가한다는 것을 기억해야 할 것이다. 만약, 분기를 해야할 경우는 많지만, 연속적인 값이 아닐 경우에는 배열을 연결리스트 형식으로 만들어서 함수의 포인터를 저장해야 할 메모리 크기를 줄여줄 수도 있을 것이다. 가능하면 분기를 많이 하기보다는 바로 접근할 수 있는 방법을 찾는 것이 좋다.

09. 가능성성이 높은 조건문을 먼저 실행한다.

; 가능성이 높은 조건을 먼저 검사하는 것이 조금이라도 실행 시간을 줄일 수 있다. 따라서, 자주 생길 것 같지 않은 조건은 나중으로 미뤄서 점검한다. 물론, 최대 지연시간은 동일할지 몰라도 전체적인 지연시간은 줄어들 것이다. 각각의 조건에 대한 실행 가능성이 다를 경우, 높은 가능성을 가지는 것을 먼저 비교하는 것이 좋다. 특히, 가능성의 거의 없지만 발생할 수 있다고 생각하는 조건문은 나중에 처리하도록 한다.

10. 빨리 복귀(Return) 하도록 한다.

; 함수가 빨리 처리를 완료할 수 있도록 만들기 위해서는, 해당하는 경우만 처리한 후 곧바로 복귀하는 것이 성능에는 도움이 된다. 물론, 이럴 경우 탈출구(EXIT Point)가 늘어나고, 주의하지 않을 경우 자원

의 획득이나 할당이 복구되지 않을 가능성이 있다. 과거에는 하나의 함수에 하나의 복귀 지점을 두는 것이 일반적이었으나 최근에는 빨리 처리를 마치고 복귀하는 것이 더 적은 코드를 작성하고 성능 향상에도 도움이 된다고 생각되고 있다.

11. “goto”를 적절하게 사용한다.

; 구조가 복잡해질 가능성이 있는 경우에는 “goto”를 이용해서 빨리 빠져 나오도록 한다. 물론, 자원의 할당이나 상태 변화가 있었을 경우에는 그것도 처리해야 한다. 그리고, “goto”는 제어(흐름, 실행) 순서를 역으로 거슬러가선 안되며, 반드시 앞으로(Forward Jump)만 가는 것이 좋다. 될 수 있으면 가는 곳의 위치와 멀리 떨어지지 않도록 만들어 주어야 할 것이다. 즉, 중첩된 블록을 많이 건너뛸 수록 이해하기 어려운 코드가 만들어질 가능성이 높다. 처리량을 줄일 수 있다는 점에서 “goto”를 사용하지 않을 이유는 없으며, 될 수 있으면 명확하게 보이도록 사용해야 한다.

12. 함수의 파라미터와 복귀 값을 CPU의 아키텍처에 맞춘다.

; 파라미터 값을 전달 받거나 복귀 값을 전달하기 위해서 컴파일러가 자동으로 변수의 타입을 변환하기 위해서 부가적인 일을 하지 않도록 만들어 주기 위함이다. 눈에 보이지는 않지만 컴파일러는 변수들을 자신이 다루기 쉬운 형태로 변환하는 경우가 많기에, 크기가 정말 중요한 경우를 제외하고는 기본적으로 정해진 부호 없는 정수 크기의 형을 사용하는 것이 좋다. 컴파일러는 타입이 다른 경우 자동으로 변환을 위한 코드를 삽입하기에 필요없는 오버헤드를 줄이기 위해서도 필요한 일이다. 또한, 자신의 CPU 레지스터 크기에 맞는 변수를 가장 최적화를 잘 할 수 있기에 이를 위해서도 필요하다.

13. 함수 내의 변수들을 CPU의 아키텍처에 맞춘다.

; 대부분의 변수는 항상 CPU 아키텍처에서 제공하는 정수 변수의 크기를 따르는 것이 가장 효과적이다. 추가적인 라이브러리의 사용이나 변환의 발생을 미연에 방지하는 것이다. 일반적으로 개발자들은 자료 구조에 적당한 크기의 변수를 선언하는 경향이 있지만, 처리의 효율을 따진다면 CPU 아키텍처의 레지스터 크기에 지역변수를 맞추는 것이 좋다. 어차피 레지스터는 나누어서 할당되지 않는다.

14. 배열의 인덱스 계산을 줄인다.

; 배열에 접근하기 위해서는 항상 인덱스와 저장된 값의 형의 크기를 곱해야 한다. 따라서, 이런 것들을 줄여주면 더 효과적으로 배열을 사용할 수 있다. 곱셈 연산은 가능한 줄이는 것이 좋다. 배열은 저장된 원소의 크기에 인덱스의 곱과 첫 번째 배열 원소의 주소 값을 합해서 접근한다. 따라서, 최소한 곱셈 한번과 더 셈이 한번 일어나야 하나의 원소에 접근할 수 있다. 또한, 배열로 저장된 원소는 순차적으로 접근하는 경우가 많으므로 적절한 방법으로 곱셈을 줄여줄 가능성이 높다. 추가적으로 배열이 다차원인 경우에 비해서 1차원 배열을 연속해서 접근하는 것이 성능 향상에 도움이 된다.

15. 포인터로 여러 단계를 거쳐야만 접근하는 값의 사용을 줄인다.

; 포인터를 여러 번 따라가서 접근 해야하는 변수는 주소 계산과 메모리 접근이 자주 발생하게 된다. 따라서, 미리 계산된 중간 포인터의 값을 기억하고 있다가 바로 접근하거나, 혹은 임시 변수에 값을 읽어서 재활용하는 것이 좋다. 주소 접근의 연산도 마찬가지로 곱셈과 덧셈, 메모리 접근으로 이루어져 있기에, 이런 부분들을 줄여주면 성능 개선의 여지가 높다. 특히, 배열과 마찬가지로 반복문 내에서 처리되는 경우에는 더 효과가 클 것이다.

16. 전역 변수를 사용하더라도 한번만 값을 얻어오도록 만든다.

; 전역 변수는 될 수 있으면 사용하지 말아야 하지만, 어쩔 수 없이 사용해야 하는 경우라면 접근 회수를 줄여주는 것이 좋다. 함수를 호출할 때 함수의 내부에서 전역 변수를 접근하지 않도록 복사된 값을 넘겨주는 것도 한가지 방법이다. 전역 변수는 항상 메모리를 접근해야 하기에 레지스터에 할당되는 지역 변수보다 접근 오버헤드가 크다. 물론, 지역 변수라고 하더라도 너무 큰 변수이거나 개수가 많아지면, 메모리 접근을 해야 한다. 하지만, 일반적으로 전역 변수를 사용하는 것은 코드의 복잡도와 성능에 악영향을 준다고 생각해야 한다.

17. 메모리 복제 회수를 줄인다.

; 메모리 이동은 오버헤드가 큰 연산이다. 즉, 메모리에 접근을 최소한 두 번은 해야하기 때문이다. 따라서, 가능하면 메모리에서 가져온 데이터를 재활용해서 사용할 수 있도록 해야 한다. 포인터를 사용하는 것도 좋은 방법이다. 하지만, 동기화 문제가 발생할 수 있기에, 데이터 값의 유효성을 잘 관리해 주어야 한다. 임베디스 시스템의 CPU를 설계한다면, 메모리 복제 회수를 줄이기 위해서 메모리 이동 연산을 하드웨어로 처리하는 경우도 있다. 만약, 그것이 지원된다면 고려해보는 것이 좋을 것이다.

18. 분리된 함수 들이 서로 관련이 있다면 합치도록 한다.

; 자주 있는 경우는 아니지만, 두 개의 함수 간에 하는 일이 관련성이 강하다면 같이 묶어서 하나의 함수로 만들어주는 것이 컴파일러의 최적화를 도울 수 있다. 하지만, 지나치게 긴 함수는 오히려 컴파일러가 최적화를 못하도록 만들 수 있으므로 주의해야 한다. 만약, 작게 만들어진 몇 개의 함수가 관련성이 강하고 병목지점이라고 확인된다면, 그 함수들을 다 묶어서 하나의 긴 함수를 만들어서 최적화를 고려해 보는 것이 좋다. 물론, 이런 경우에는 반드시 코드의 가독성을 고려한 판단이 필요하며, 상황에 의존적이므로, 먼저 함수의 길이를 짧게 유지하려는 노력이 선행되어야 할 것이다.

19. 항상 고정된 입력에 고정된 값을 주는 것은 테이블(배열)로 대체한다.

; 입력이 고정되어 있고 그에 따른 출력이나 행동이 정해져 있다면, 테이블을 이용해서 중간 계산 과정이나 조건문을 이용한 분기를 줄여줄 수 있다. 예를 들어, 특정 값을 읽어서 복잡한 계산을 한 후에, 특정한 값을 돌려주어야 할 경우, 그냥 테이블을 이용해서 처리하면 대부분의 계산 시간을 줄일 수 있다. 하드 코딩이라 그렇게 만든 이유를 자세히 설명할 필요는 있을 것이다. 예를 들어, 특히 복잡한 알고리즘을 이용해서 중간 계산을 실행해야 하는 경우, 고정된 입력에 결과값이 항상 동일하다고 판단된다면, 과감하게 테이블로 계산을 대체하도록 한다.

20. 영향을 크게 주지만 작은 부분의 코드를 찾아서 기계어(Assembly)로 작성한다.

; 마지막으로 생각해 볼 수 있는 방법이다. 만약, 특정 부분을 기계어로 구현 했다면, 이식성(Portability) 문제로 인해서 CPU가 달라질 경우 제대로 동작하지 않을 수도 있다. 그리고, 유지 보수가 어렵기 때문에 가능한 제일 성능에 민감한 부분을 찾아, 한정된 부분만 기계어로 구현해야 한다. 특히, 기계어는 컴파일러나 실행 환경에 밀접한 관련이 있기에, 분리된 파일로 따로 관리하는 것이 좋을 것이다. 하드웨어에 의존적이지 않은 코드와 같이 관리한다면 인터페이스의 분리가 어렵기 때문이다.

항상 최적화를 할 때는 가장 먼저 구조적인 개선(전체 최적화) 방법을 찾아야 한다. 구조 개선은 150% 이상의 성능 향상도 가능하지만, 나쁜 구조를 그냥 두고 지역적인 최적화를 해서는 20%도 높이기 힘들다. 따라서, 큰 것을 먼저하고 작은 것들은 차근차근 정복해 나가는 방법을 사용하는 것이 좋다. 여기서 이야기 한 것 이외에도 다양한 것들이 있으니, 사용하는 컴파일러에 대한 메뉴얼을 자세히 읽어볼 것을 권장한다.

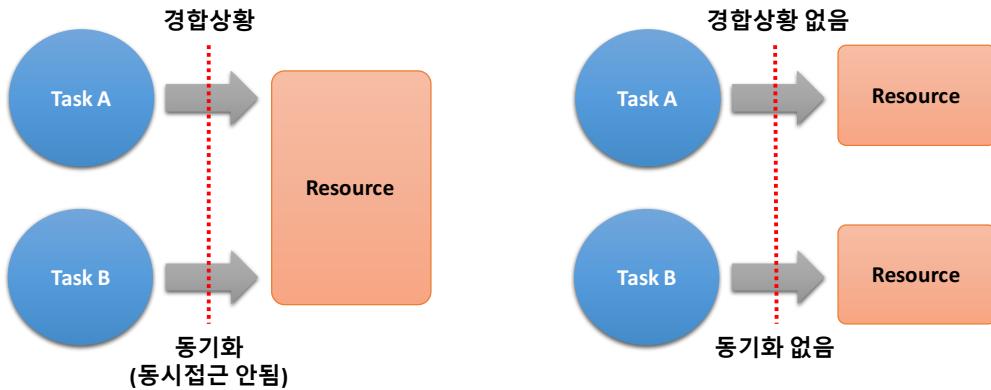
우려할 수 있는 문제점은 구조를 변경해서 버그가 크게 늘어날지도 모른다는 불안감이다. 이미 구조화가 안된 코드가 과제 완료까지 얼마남지 않은 상황이라면 섣불리 구조화를 시도 해서는 안된다. 이때는 그냥 과제 완료 때까지 그냥 두거나 지역적인 최적화에만 힘을 쏟아야 할 것이다. 구조화 자체가 큰 일이 될 수 있기에 별도의 과제로 만들어서 진행하는 것이 효과적일 것이다. 섣부른 구조화는 안고치는 것보다 못한 결과를 낳을 수 있으며, 그것으로 인해 개발자에게 잘못된 인상을 줄 수 있기 때문이다.

[주의]

앞으로 볼 예제 들에서 지역변수의 크기를 잡는 경우를 종종 볼 수 있는데, 실제 코딩에서는 사용해선 안된다. 될 수 있으면 지역 변수의 개수도 적게 해야하며 지역변수의 크기도 줄여주는 것이 좋다. 사용하는 시스템마다 기본적으로 설정되는 스택의 크기가 있기에 예제가 동작하지 않을 수도 있다. 이때는 지역변수의 크기를 줄여보기 바란다. 물론, 기본으로 설정된 스택의 크기는 수정이 가능하다.

[개별적으로 동작 하면 문제가 없지만 동시에 동작 하면 성능 문제가 생길 수 있다.]

자원을 공유할 때 흔히 생기는 문제다. 시스템의 자원이란 CPU, 메모리, DMA 채널, 실행을 동기화 해야 할 부분 등등 다양한 것들이 있을 수 있다. 공유해야 하는 자원이 하드웨어로 연결된 경우에는 큰 문제가 될 수 있다. 만약, 다른 두 개의 태스크(Task)가 동시에 접근하는 경우 데이터가 깨지거나, 혹은 제대로 된 값을 읽지 못할 수 있다.



이때는 두 개의 태스크(일반적으로 실시간 시스템에서 사용하는 스케줄링의 최소단위) 간에 컨텍스트 스위칭(Context Switching)이 발생하지 않도록 만들어, 혼자만 독점적으로 자원을 사용할 수 있도록 만들면 문제는 해결된다. 하지만, 근본적인 성능 향상은 이룰 수 없다. 이런 경우를 미리 하드웨어 설계 단계에서 알았다면 공용 자원에 대한 접근 방법을 미리 설계에 반영 했을 것이다. 가능하면 공유되는 하드웨어 자원에 대해서는 사용하는 방법을 미리 확인해야 한다.

운영체제를 사용하지 않고 개발하는 경우에는 동기화 문제에 대해서 인터럽트와 관련지어 생각해야 한다. 즉, 자원을 사용하는 소프트웨어가 동작하고 있을 때, 인터럽트 발생으로 인해서 공유 자원에 대한 접근이 발생한다면 무결성(Integrity)이 깨질 가능성이 있다. 따라서, 이때는 자원을 접근하는 순간 인터럽트가 발생하지 못하도록 만들어야 하며, 자원 사용이 끝났을 때 다시 인터럽트가 발생할 수 있도록 만들어 주어야 한다. 그리고, 가능한 인터럽트가 발생하지 못하는 구간을 짧게 유지하는 것이 좋다. 그렇지 않다면, 인터럽트에 대한 처리률(Throughput)이 크게 낮아질 것이다. 즉, 시스템의 외부 이벤트에 대한 반응 속도가 낮아진다.

[전역 변수 사용 줄이기]

전역 변수는 코드의 이해를 어렵게 만들 뿐 아니라 최적화에도 도움이 되지 않는다. 전역변수는 함수나 포인터를 이용한 접근 등등 다양한 방법으로 언제 어디서든 사용될 수 있기에, 레지스터(Register)를 이용해서 데이터를 처리할 수 없다. 따라서, 컴파일러의 입장에서는 이런 전역변수를 계속 메모리에서 읽어와야 할 필요가 생기게 되며, 처리가 끝나면 다시 메모리로 보관해야 한다. 이런 상황이라면 전역 변수를 반복문과 같은 곳에서 사용하는 것이 성능 저하를 일으킬 가능성이 높다.

만약, 전역변수가 다른 곳에서 동시에 사용될 가능성이 없다면, 한 번만 읽어와서 지역변수에 저장하고 사용하는 것도 고려해 볼 만하다. 즉, 지역변수는 레지스터에 할당될 가능성이 높으며, 지역변수를 사용하는 것이 반복문 안에서의 직접적인 메모리에 대한 읽고 쓰기 동작을 줄여줄 것이기 때문이다.

원칙은 전역변수를 될 수 있으면 적게 사용하는 것이 좋다는 것이다. 전역변수가 늘어나면 그것으로 인해서 생기는 문제점도 같이 늘어날 것이다. 문제는 원인을 제거하는 것이 좋지, 원인은 두고 현상만 쫓아가면서 해결하는 것은 근본적인 해결책이 아니다.

```
int global_variable;
```

```
void function( void ) {
    for( int i = 0; i < 1000; i++ )
    {
        global_variable += i;
    }
}
```

.....

```
void function( void ) {
    /* Critical Section Enter */
    temp_global = global_variable;
    /* Critical Section Exit */
    for( int i = 0; i < 1000; i++ ) {
        temp_global += i;
    }
    /* Critical Section Enter */
    global_variable = temp_global;
    /* Critical Section Exit */
}
```

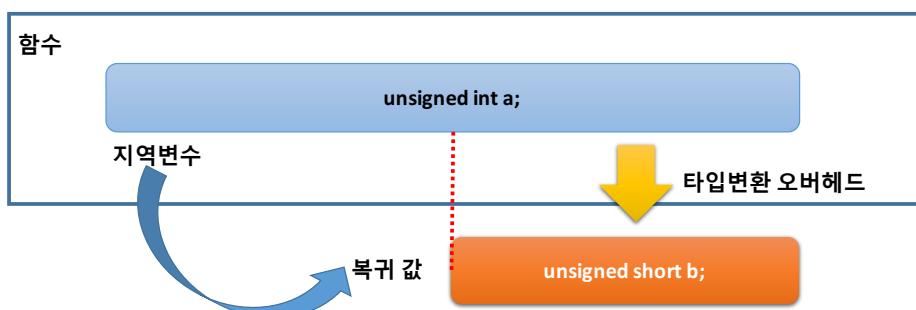
위의 예는 전역변수를 지역변수에 읽어서 사용하는 경우를 보여준다. 여기서 한 가지 짚고 넘어가야 할 부분은 전역변수의 값이 다른 곳에서도 변경이 발생할 수 있기에, 전역변수와 관련된 부분을 "CriticalSection"처럼 보호해 주어야 할 필요도 있다. 리얼타임 운영체제(RTOS)를 사용하는 임베디드 시스템 같은 경우에는 인터럽트나 컨택스트 스위칭(Context Switching)이 발생하지 않도록 해주어야 한다.

컴파일러의 경우 반복문에 대한 효율을 높이는 방향으로 최적화 기법을 많이 지원한다. 따라서, 전역 변수를 사용하게 되면 컴파일러 자체가 최적화를 포기할 수 밖에 없는 상황을 만드는 것과 같다. 즉, 메모리를 항상 읽어서 변화된 값을 확인해야 하기 때문이다. 따라서, 명령문의 순서를 바꾸는 것도 전역 변수 값에 영향을 받을 수 있기에 최적화를 할 수 없다.

[모든 변수는 기본적으로 정수로 선언하는 것이 좋다.]

변수의 크기를 적절한 크기로 맞추는 것이 반드시 성능을 좋게 보장하는 것은 아니다. 주로 사용하는 CPU가 지원하는 크기로 맞추는 것이 최적의 성능을 발휘할 가능성이 높다. 만약, 변수의 타입이 여러가지가 섞여서 사용된다면 상황은 더 좋지 않다. 즉, 타입 변환을 위한 코드들이 컴파일러에 의해서 자동으로 추가되기 때문이다.

사용되는 메모리를 줄이려고 작은 크기의 변수에는 "short" 혹은 "char"로 타입(Type)을 주게되면, 컴파일러는 다른 타입의 변수에 저장하기 위해서 어셈블리 명령어를 몇 개 더 사용해야 한다. ARM과 같은 RISC CPU에서는 이런 부분들이 라이브러리 코드 처럼 추가되어 오버헤드로 작용한다. 따라서, 가능하면 모든 변수들은 정수형으로 기본 선언하는 것이 좋다.



특히, 디바이스 드라이버나 펌웨어(Firmware)와 같은 경우에는 거의 정수 밖에 사용하지 않는다. 레지스터나 기타 장치들에 대해서 쓰기나 읽기를 할 때는 항상 정수만 사용한다. 따라서, 드라이버와 같은 코드에서는 “Floating Point” 변수나 “Short”, “Char” 타입 등의 변수는 별로 사용할 일이 없을 것이다. 함수의 경우에도 어차피 스택 공간에서 변수의 메모리 공간을 사용하거나, 혹은 레지스터에서 변수를 사용하기에, 타입을 최적화 시킨다고 크게 도움이 안된다. 스택도 “32bit”이나 “64bit” 단위로 사용할 것이기 때문이다.

정수 연산이 CPU에서는 가장 빠르다. 즉, CPU의 레지스터 크기에 맞는 가장 적합한 크기의 변수가 정수다. 따라서, 다른 변수 타입에 비해서 연산 속도가 빠를 수 밖에 없다. 특히, 부호를 가진 정수 보다는 부호가 없는 정수가 더 빠르다. 부호를 가진 경우에는 부호에 대한 처리를 위해서 하드웨어가 할 일이 더 많아지기 때문이다. 따라서, 가능한 모든 변수는 부호가 없는 정수로 선언하는 것이 좋다. 변수의 크기를 최적화하기 위해서 딱 맞는 정도의 변수 타입을 선언해서 쓰는 것은, 형 변환(Type Conversion)을 자동으로 발생시킬 가능성을 높여 성능을 저하 시키게 된다.

[참고]

“Floating Point”연산의 경우 ARM에서는 기본으로 라이브러리(Library)를 통해서 소프트웨어적으로 제공한다. 따라서, “Floating Point” 변수가 선언되면, 필요한 라이브러리들이 자동으로 추가 되기에 성능의 저하와 동반해서 크기의 증가도 발생한다. “Floating Point”가 정말 필요한지를 따져봐야 하고, 만약 정말 필요하다면 오히려 소수점 이하 얼마 까지 정밀도를 가져야 하는지 검토한 후에, 정수로 만들어서 사용할 수도 있다. 예를 들어, 소수점 이하 두 자리 까지가 유효하면, 100을 곱해서 정수로 만들어 주고, 연산 결과의 최종본은 나중에 100으로 다시 나누어 주면 된다. 물론, 유효 수자의 범위가 중요하기 주의해서 해야할 것이다.

[참고]

GCC를 이용하고 Eclipse에서 개발을 하고 있다면, "Project --> Properties --> C/C++ Build --> Settings --> GCC C Compiler --> Miscellaneous"에 "-Wa,-aIn='\${CWD}\\${InputFileName}.cod'"를 설정해서 Assembly 코드와 C코드를 같이 보도록 하자. 코드를 같이 보면 최적화를 했을 때 발생하는 실제 실행될 기계어 코드가 어떻게 만들어지는지 확인해야 한다.

[코드의 실행 시간 구하기]

함수를 호출하고 알고리즘을 실행하는 시간을 구하기 위해서는 몇 가지 함수를 이용해야 한다. 사용하는 개발환경에 따라 차이는 있지만, 오래된 함수인 "gettimeofday()"는 다 제공하고 있기에, 일단 우리는 이 함수를 이용해서 실행 시간을 측정하도록 하겠다(최근에는 "clock_xxx()"와 같은 함수를 사용하는 것이 좋다고는 하지만, MinGW에서는 아직 제공되지 않는다.)

"gettimeofday()"함수를 사용하기 위해서는 "struct timeval"라는 구조체를 사용하는데, 간단히 초(Second)와 마이크로 초(Microsecond)를 나타내는 tv_sec와 tv_usec 필드로 구성되어 있다.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

unsigned long fibonacci(int nth) {
    if ((nth == 0) || (nth == 1)) {
        return 1;
    }
    return fibonacci(nth - 2) + fibonacci(nth - 1);
}
```

```

int main(void) {
    puts("This is a clock tick calcuation program!!!\n");
    unsigned long result = 0;
    struct timeval start;
    struct timeval stop;
    struct timeval elapsed;

    /* 실행 시간을 구한다. */
    gettimeofday(&start, NULL);
    result = fibonacci(40);
    gettimeofday(&stop, NULL);

    /* 실행 시간을 표시하기 위해서 초와 마이크로 초를 조정한다. */
    elapsed.tv_sec = stop.tv_sec - start.tv_sec;
    if (stop.tv_usec < start.tv_usec) {
        elapsed.tv_sec--;
        elapsed.tv_usec = (1000000 - start.tv_usec) + stop.tv_usec;
    } else {
        elapsed.tv_usec = stop.tv_usec - start.tv_usec;
    }

    printf("To calculate fibonacci number, it takes %lu secs and %lu usec.\n",
           elapsed.tv_sec, elapsed.tv_usec);
    printf("The fibonacci value is : %lu\n", result);

    return EXIT_SUCCESS;
}

```

간단히 하기 위해서 “main()”에서 호출하는 구조를 취했지만, 필요하다면 함수로 따로 정의해서 호출 하면될 것이다. 앞의 예는 피보나치 수를 구하는 함수의 실행 시간을 구하는 것이다. 의미있는 시간의 변화를 확인하기 위해서 10보다는 큰 순서에 있는 피보나치 수를 구하고 있다. 너무 작은 순서의 피보나치 수를 구하기 위해서는 좀 더 정밀한 시간을 구할 수 있는 함수를 만들어야 할지도 모른다.

실제로 “gettimeofday()” 생각보다 정확한 함수가 아니다. 최근의 CPU들은 “GHz”단위로 동작하기에 “Microsecond”단위로 구하는 것이 큰 의미를 지니지 못할수도 있다. 따라서, 실행 시간을 정밀하게 측정하기 위해서는 “Nanosecond”단위까지 측정할 수 있는 방법이 필요하다.

[정밀한 실행 시간 얻기]

앞에서 본 실행 시간을 구하는 것은 시스템 호출(System Call)을 사용하기에, 운영체제가 무슨 일을 하고 있느냐에 따라 영향을 받을 수 있다. 또한, 구한 값의 정확도도 낮아서 제대로 된 결과라고 믿을 수 없다. 더 정확하고 고해상도의 값을 구하기 위해서는 하드웨어에서 직접 값을 읽어와야 할 필요가 있다. “x86 “계열의 CPU를 사용한다면, “rdtsc(Read Time Stamp Counter)”라는 명령어를 이용해서 이와 같은 일을 할 수 있다.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

#if defined(__GNUC__) && (defined(__MINGW32__) || defined( __MING64__))

```

```

/* GNU C 컴파일러를 사용하는 경우 */
static inline uint64_t get_cpu_cycle(void) { /* GNU C 컴파일러의 인라인 함수 선언 */
    uint64_t ticks;

    __asm volatile("rdtsc" : "=A"(ticks));
    return ticks;
}

#ifndef _MSC_VER
/* Windows에서 Visual Studio를 사용하는 경우 */
#include <intrin.h>
#pragma intrinsic(_rdtsc)
static __inline uint64_t get_cpu_cycle(void) { /* Windows 인라인 함수로 선언 */
    return _rdtsc();
}
#endif

```

사용하는 컴파일러나 실행 환경에 따라 사용할 수 있는 어셈블리 명령어가 다를 수 있다. 또한, 미리 제공되는 라이브러리를 사용할 수도 있기에, 환경을 읽어서 컴파일러에서 처리할 수 있도록 해주었다. GNU 컴파일러를 사용한다면, 인라인 어셈블리를 사용해서 처리하도록 만들었다. 윈도우에서 Visual Studio를 사용하고 있다면, 동일한 역할을 하는 함수로 “`_rdtsc()`”를 제공하고 있다. 이 함수의 출력값은 “64bit” 부호 없는 정수(`uint64_t`) 형태를 가진다. 정수의 타입 정의는 “`<stdint.h>`”를 따른다.

[참고]

앞에서 본 실행 시간을 측정하는 함수의 코드는 나중에 실행 시간을 측정하는 코드에서 재활용 된다. 이후에는 지면상 생략하도록 하겠다. 별도의 파일로 만들어서 재활용 하면 된다.

```

int decimal_convert(char* binary) {
    unsigned int size = 0;
    unsigned int value = 0;
    int i, j;

    /* Check input binary */
    if (binary == NULL) {
        printf("Cannot convert NULL binary pointer!!!\n");
        return -1;
    }

    /* Get the size of binary value array */
    size = strlen(binary);

    /* Convert binary into decimal */
    for (i = size - 1, j = 0; i >= 0; i--, j++) {
        if (binary[i] == '1') {
            value += 1 << j;
        } else if (binary[i] == '0') {
            /* Do nothing */
            continue;
        } else {
            /* Not a binary value */
            return -1;
        }
    }
}

```

```

    }
    return value;
}

```

앞의 “decimal_convert()”함수는 실행 시간을 알고 싶어하는 대상 코드다. 입력받은 문자열로 표현된 이진값을 십진수로 변환하는 일을 수행한다.

```

int main(void) {
    puts("Binary To Decimal Convert Example 01");
    char *input[] = { "1101", "1", "0", "01", "10", "11", "0101", "11011011", "3101" };
    unsigned int size = sizeof(input) / sizeof(input[0]);
    unsigned int i;
    uint64_t before, after, counter;

    before = get_cpu_cycle();
    printf("The time tick counter : %I64d\n", before);
    for (i = 0; i < size; i++) {
        printf("The decimal value of %12s --> %5d\n", input[i],
               decimal_convert(input[i]));
    }
    after = get_cpu_cycle();
    printf("The time tick counter : %I64d\n", after);
    counter = after - before;
    printf("The time tick counter : %I64d\n", counter);
    printf("Done!!!\n");
    return EXIT_SUCCESS;
}

```

측정하고자 하는 함수의 실행 시간을 알기 위해서는 시작전에 CPU의 타임 스탬프(Time Stamp)를 읽어서 보관하고, 끝났을 때 다시 타임 스탬프 값을 읽어서 차이를 구해야 한다. 물론, 이렇게 만들어진 값은 “초”단위가 아니다. 그 값을 알기 위해서는 CPU의 동작 주파수로 나누어 주어야 실행된 시간을 알 수 있다. 아래의 코드는 “Mac OS X”에서 “Eclipse”에서 측정하기 위한 방법을 추가한 것이다 (“__APPLE__”). “Mac OS” 자체의 컴파일러는 GNU 컴파일러와 유사하기에 동일한 인라인 어셈블리를 사용할 수 있다. “Mac OS”的 경우에는 “64bit”값의 정수를 출력하기 위해서 “printf()”함수에서 사용하는 포맷터(Formatter:출력 포맷을 가르키는 문자열)가 다르다는 점은 기억하기 바란다.

```

#if defined(__GNUC__) && (defined(__MINGW32__) || defined( __MING64__))
/* Windows의 Eclipse에서 GNU Compiler를 이용하는 경우 */
static inline uint64_t get_cpu_cycle(void) {
    uint64_t ticks;

    __asm volatile("rdtsc" : "=A"(ticks));
    return ticks;
}
#elif defined(_MSC_BUILD)
/* Windows Visual Studio의 경우 */
#include <intrin.h>
#pragma intrinsic( __rdtsc)
static __inline uint64_t get_cpu_cycle(void) {
    return __rdtsc();
}

```

```

#ifndef __APPLE__
/* Mac OS X Eclipse의 경우 */
static inline uint64_t get_cpu_cycle(void) {
    uint64_t ticks;

    __asm volatile("rdtsc" : "=A"(ticks));
    return ticks;
}
#endif

/* CPU Tick Count를 출력하는 함수 들 */
void print_tick_count(uint64_t input) {
#if defined(__GNUC__) && (defined(__MINGW32__) || defined( __MING64__))
    printf("The time tick counter : %I64d\n", input);
#elif defined(_MSC_BUILD)
    printf("The time tick counter : %I64d\n", input);
#elif defined(__APPLE__)
    printf("The time tick counter : %" PRIu64 "\n", input);
#endif
}

```

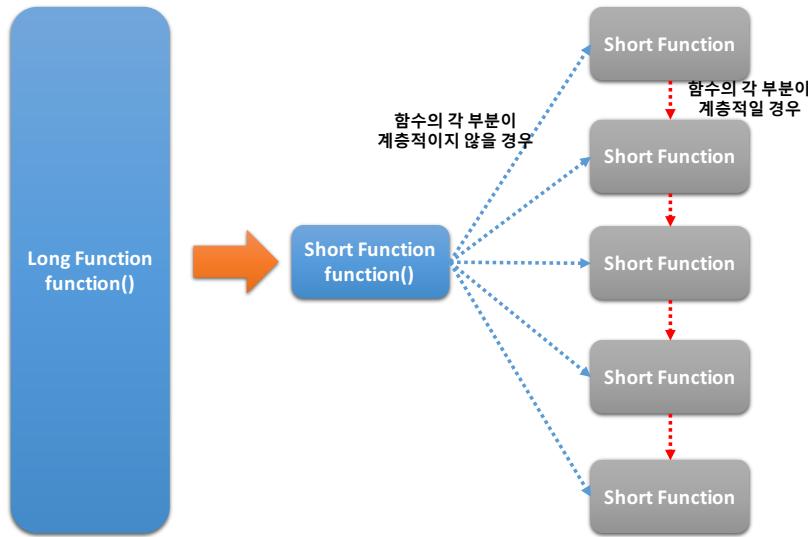
ARM과 같은 CPU를 사용하는 경우에는 “CCNT(Cycle Counter)” 레지스터를 이용해서 같은 일을 할 수 있다. 하지만, 사용하기 위해서는 특권 명령어를 사용해서 미리 시스템 초기화 시에 설정해 주어야 하는 일이 있다. 설정은 부트로더(Boot Loader)나 커널(Kernel)에서 초기화 시에 해주어야 한다. 관련된 사용 방법은 인터넷에 있으니 찾아서 직접 해보기 바란다. 참고로, CPU별로 성능을 측정하기 위한 방법들로 어떤 것들이 있는지 파악한 후에, “get_cpu_cycle()”과 같은 함수를 만들어야 할 것이다.

한가지 주의할 것은 이렇게 구한 시간이 완전히 믿을만 한 것은 못된다는 사실이다. 인터럽트가 활성화된 상황에서는 실행 중에 컨텍스트 스위칭(Context Switching)이나 인터럽트 서비스로 인해서 추가된 시간이 있을지도 모른다. 정말 정확한 시간을 알고자 한다면, 시스템의 콜에 따른 오버헤드와 인터럽트 및 운영체제의 개입에 대한 것도 제거해야 할 것이다. 따라서, 일반적인 PC환경에서는 어느 정도 오차가 있다고 가정해야 한다. 또 한가지 고려할 상황은 하드웨어의 속도가 빠를 경우에는 미미한 차이를 보일 수도 있지만, 낮은 사양에서는 눈에 띄는 차이를 보일 수도 있다는 점이다.

결과적으로 한번 측정한 결과를 믿어서는 안되며, 될 수 있으면 여러번 측정한 후에 평균과 표준 편차를 이용해서 구간으로 측정해야 한다는 점이다. 즉, 몇 %정도가 어느 구간에 들어올 수 있는지를 찾아서 상대 비교를 하는 것이 좋을 것이다.

[함수의 호출 경로(Call Path)가 길어지지 않게 만들기]

코드를 만드는 방법은 다양하게 있을 수 있다. 예를 들어, A가 B를 호출하고, B가 C를 호출, C가 D를 호출하는 "A->B->C->D"로 만들수도 있고, 혹은 "A->B, A->C, A->D"와 같이 A가 B,C,D를 호출하는 구조로 만들 수 있다. 그럼 어떤 선택이 더 나은 선택이 될 수 있을까? 이 문제에 대한 해답은 어떤 상황에서 함수의 호출이 사용 되는지에 따라 다를 수 있다.



전자의 경우에는 B는 A와 관계를 맺으면서 C에 의존적이다. C는 다시 B와 관계가 있으면서 D에 의존적이다. 따라서, B와 C는 C와 D와의 의존성을 가지고 있다. 후자의 경우에는 B,C,D가 의존성을 가지지 않으며, A와만 관계를 맺고 있다. 당연히 호출 경로도 더 짧다. 함수 호출의 개수를 따지면, 둘 다 3번으로 동일하다. 따라서, 조금 더 의존성이 약한 코드가 코드를 재사용 할 가능성이 높으며, 수정 하기도 용이하다는 것을 알 수 있을 것이다.

물론, 하나의 기능을 구현할 때는 전자와 같이 계층상의 분리를 하는 것이 좋다. 만약, 동일한 수준의 일을 처리하고 있다면 후자의 경우가 좋다. 즉, B,C,D가 동일 계층에 놓일만큼 수준이 같은 추상화를 A에 제공하고 있어야 한다. 그렇지 않을 경우에는 A가 각각의 계층을 직접 접근하는 문제가 발생할 수 있다. 계층을 넘어서 접근하는 것은 프로그램의 구조를 복잡하게 만들 가능성성이 높기에 될 수 있으면 하지 않는 것이 좋다.

만약 그런 경우가 발생한다면 시스템의 인프라(Infra)와 같은 공용화 된 함수를 이용하는 경우에 한정하도록 해야 할 것이다. 예를 들어, 표준 라이브러리 함수나 시스템 콜과 같은 경우가 될 수 있다. 계층화 된 아키텍처를 그림으로 표현할 때, 위 아래로 길게 배치하는 모듈이 여기에 해당한다.

코드는 규형을 가지는 것이 중요하다. 즉, 특정 모듈이 지나치게 비대하다면, 그 모듈을 더 잘게 나누어야 한다는 말이다. 모듈에 코드가 골고루 분포한다는 말은 모듈의 역할과 책임 범위가 잘 분포되어 있다는 말과 동일하다. 물론, 예외적으로 라이브러리와 같이 도입한 모듈의 경우는 이에 해당하지 않는다. 그런 것들은 관리의 주체가 다르기 때문이다. 어쨌든 자신이 작성한 코드라면 적어도 모듈간의 규형은 이루는 것이 좋다. 한 디렉토리에 있는 파일의 개수가 많거나, 파일 내부의 함수의 개수, 함수의 길이, 전체 파일의 길이, 함수 내에 있는 변수의 개수 등도 균일하게 가져가는 것이 좋다.

이런 말들이 마치 물리적인 것만을 강조한다고 생각할지도 모르지만, 우리의 코드도 어차피 물리적인 구현에 구속된 것일 수 밖에 없다. 물리적으로 지나치게 비대한 코드는 이해하기 어려우며, 상세화와 추상화가 같이 공존한다면 더 이해하기 힘들다. 따라서, 계층적으로 나누고, 수평적으로도 균일하게 만드는 일은 코드의 유지보수에 영향을 주게 된다. "한 번에 한 가지 일만 잘하라"는 것이 이러한 이유 때문이다. 그리고, 그렇게 만들어진 함수들이 군집을 이뤄 파일이되고, 다시 관련된 파일들이 모여서 모듈이 된다. 한 가지 기능을 하는 것으로 통합될 수 있는 모듈들은 서브 시스템이 되고, 모든 시스템은 다양한 서브 시스템의 도움으로 동작하는 것이다.

아래의 코드는 코드를 부분들로 나누어서 “inline” 함수로 만들어준 경우다. 작은 반복(1,000번 정도)에서는 차이가 미미하지만(오히려, “inline” 함수로 만든 코드가 성능이 더 좋은 때도 있다), 반복 횟수(1,000,000)가 늘어나면 의미있는 차이가 있을 수 있다. 하지만, 최적화(-O3) 옵션을 설정한 상태에서

는 거의 비슷한 결과가 만들어질 수 있다. 함수 내에서 실행되는 코드에 의존적 이기는 하지만, 될 수 있으면 작은 함수 들의 모임으로 코드를 분해해 주는 것이, 성능에 대한 영향을 최소화하면서 읽기 쉬운 코드를 만드는 핵심이다.

```

static inline unsigned int add(unsigned int result, unsigned int value) {
    return result + value;
}

static inline unsigned int dec(unsigned int result, unsigned int value) {
    return result - value;
}

static inline unsigned int mul(unsigned int result, unsigned int value) {
    return result * value;
}

static inline unsigned int dix(unsigned int result, unsigned int value) {
    return result * value;
}

#define MAX_TIMES 1000000

int main(void) {
    puts("SmallFunction Example 01"); /* prints SmallFunction Example 01 */
    unsigned int value = 0;
    unsigned int ticks;

    /* 긴 코드를 만드는 예 */
    ticks = get_cpu_cycle();
    for (unsigned int i = 1; i < MAX_TIMES; i++) {
        value = value + i;
        value = value * i;
        value = value - i;
        value = value / i;
    }
    ticks = get_cpu_cycle() - ticks;
    print_tick_count(ticks);

    /* 짧은 함수들로 나누어준 예 */
    value = 0;
    ticks = get_cpu_cycle();
    for (unsigned int i = 1; i < MAX_TIMES; i++) {
        value = add(value, i);
        value = mul(value, i);
        value = dec(value, i);
        value = dix(value, i);
    }
    ticks = get_cpu_cycle() - ticks;
    print_tick_count(ticks);

    return EXIT_SUCCESS;
}

```

}

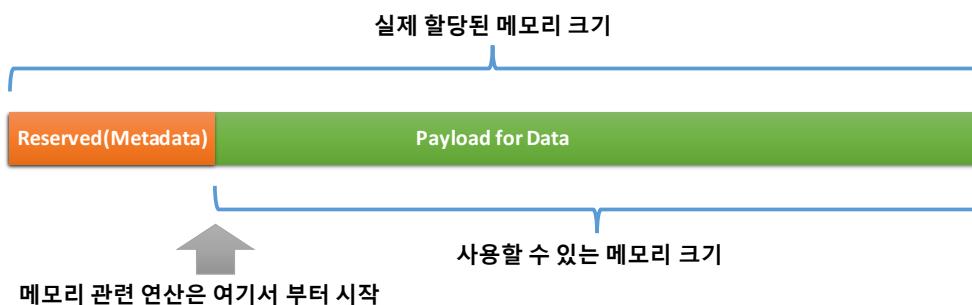
따라서, 긴 함수를 여러 개로 나누어서 “*inline*” 함수로 만드는 것은 큰 부담이 되지 안된다. 물론, 그 정도의 차이도 용납하지 않는다면, 성능이 문제가 되는 부분의 함수만 길게 만들 수도 있을 것이다. 중요한 것은 위의 코드에서 보듯이, 최적화를 위해서 긴 함수를 만든다면, 코드의 가독성은 떨어지게 된다. 오히려 “*inline*” 함수를 사용해서 긴 함수를 작은 여러 개의 조각으로 나누어서(추상화 시켜서) 보는 것이 가독성은 더 높다.

[주의]

결과는 사용하는 컴파일러, 코드가 하는 일, 최적화 옵션, 사용하는 운영체제 환경, CPU, 메모리 공간 등등 다양한 부분에 영향을 받을 수 있다. 따라서, 무조건 특정 방법을 사용하기보다는 직접 실험해 보고 결정하는 것이 좋다. 앞으로 보는 예제들은 코드를 짧게하기 위해서 비교하는 코드를 같이 실행하고 있지만, 나중에 실행되는 코드가 앞의 코드로 인해서 영향을 받을 수 있다(운영체제의 스케줄링 우선 순위 변동). 따라서, 실제 구현할 때는 비교되는 두 코드를 나누어서 별도의 실행 파일로 만들어 실행시켜야 할 것이다.

[메모리 복사 줄이기]

메모리를 접근하는 것은 레지스터를 접근하는 것보다 항상 비용이 크다. 특히, 배열이나 할당 받은 메모리 블록에 대한 초기화나 특정 값으로 설정하는 것은 성능에 영향을 줄 수 있다. 따라서, 가능하다면 메모리 복사보다는 미리 할당된 메모리를 계속 사용하는 것이 좋다.



예를 들어, 포인터를 이용해서 미리 할당된 메모리를 재사용 하는 경우가 대표적이다. 리눅스의 커널과 같은 경우에는 네트워크 패킷을 다루기 위해서 특별한 자료구조를 사용해서 메모리 복제를 최소화 시킨다. 그래픽과 같은 경우 벡터(Vector) 그래픽이 아닌 비트맵(Bitmap)을 이용할 때, 메모리 복제가 흔히 일어나게 되며, 이때는 특별한 하드웨어의 도움을 얻는 것이 더 효율적일 수 있다.

즉, 칩(Chip)을 설계할 때 전용 “메모리에서 메모리로 데이터를 옮기는 DMA(Memory-to-Memory DMA)”를 구현하는 경우도 있으며, 이 때는 단순히 옮겨야 할 메모리의 시작주소와 그 크기만 알려준 후 DMA의 동작을 구동 시켜주면 된다. 메모리 복사가 끝나면 인터럽트를 통해서 전송 완료 상태를 알려줄 것이다.

메모리 복제를 줄이기 위해서는 메모리를 가리키는 포인터를 공유하는 방법을 사용하거나, 미리 할당된 메모리를 크기별로 나누어서 시작 부분의 주소와 몇 번째 인지를 알려주는 경우도 있다. 마치 배열을 사용 하듯이, 특정 크기별로 미리 할당된 메모리를 사용한다면, 할당에 대한 운영체제의 오버헤드를 제거 할 수 있으며, 시스템에 동적인 상황을 줄여 버그 발생 가능성도 낮출 수 있다. 물론, 미리 할당된 메모리 공간에 대한 관리는 응용프로그램의 책임이 되며, 할당된 이후에는 다른 응용프로그램에서 사용할 수 없다는 점도 주의해야 한다.

[전역 변수와 지역변수의 비교]

전역 변수는 어떤 코드에서도 갑자기 바꿀 수 있기에 컴파일러는 최적화를 하지 못한다. 따라서, 최적화 옵션을 높여서 컴파일을 하더라도 성능은 크게 개선되지 못한다. 하지만, 함수 내부의 지역 변수는 다르다. 컴파일러는 지역 변수를 만나면 결과값에 영향을 주지 않는 범위에서 최적화를 하게 된다.

```
#define MAX_LENGTH 1000
unsigned int global[MAX_LENGTH][MAX_LENGTH]; /* 8(64bit) x 1000 x 1000 = 8MB */

int main(void) {
    puts("GlobalVariable Example 01");
    unsigned int local[MAX_LENGTH][MAX_LENGTH]; /* 지역 변수의 크기가 크다. */
    unsigned int ticks;

    ticks = get_cpu_cycle();
    for (unsigned int i = 0; i < MAX_LENGTH; i++) {
        for (unsigned int j = 0; j < MAX_LENGTH; j++) {
            global[i][j] = i * j;
        }
    }
    ticks = get_cpu_cycle() - ticks;
    print_tick_count(ticks);

    ticks = get_cpu_cycle();
    for (unsigned int i = 0; i < MAX_LENGTH; i++) {
        for (unsigned int j = 0; j < MAX_LENGTH; j++) {
            local[i][j] = i * j;
        }
    }
    ticks = get_cpu_cycle() - ticks;
    print_tick_count(ticks);

    return EXIT_SUCCESS;
}
```

앞의 코드는 최적화 수준이 낮은 경우에는 오히려 전역변수를 사용하는 것이 더 좋은 결과를 낼 수도 있다. 하지만, 컴파일 옵션을 높여주면 지역변수를 사용하는 경우가 더 효과적인 것을 확인할 수 있을 것이다. 물론, 이것도 실행되는 코드의 복잡도와 관련이 있는 하지만, 대부분의 경우 전역 변수를 사용하는 것 보다 지역변수를 쓰는 것이 성능 향상에도 더 도움을 얻을 수 있을 것이다.

사실 위의 코드는 성능이 크게 달라지지는 않을 것이다. 즉, 전역 변수는 어차피 메모리를 접근해야 하지만, 지역 변수도 너무 커서 레지스터를 이용할 수 없다. 따라서, 지역 변수를 접근하려고 해도 마찬가지로 메모리에 직접 접근해야 한다. 하지만, 만약 지역 변수가 레지스터에 할당될 수 있으면 이야기는 극적으로 달라질 수 있다. 앞에서 본 코드를 아래와 같이 변경하고 실행하면 그 차이를 확인할 수 있을 것이다.

```
unsigned int global; /* 일반적인 변수로 선언했다. */

int main(void) {
    puts("GlobalVariable Example 01");
    unsigned int local; /* 전역변수 선언과 마찬가지로 일반적인 변수로 선언했다. */
```

```

unsigned int ticks;

ticks = get_cpu_cycle();
for (unsigned int i = 0; i < MAX_LENGTH; i++) {
    for (unsigned int j = 0; j < MAX_LENGTH; j++) {
        global = i * j;
    }
}
ticks = get_cpu_cycle() - ticks;
print_tick_count(ticks);

ticks = get_cpu_cycle();
for (unsigned int i = 0; i < MAX_LENGTH; i++) {
    for (unsigned int j = 0; j < MAX_LENGTH; j++) {
        local = i * j;
    }
}
ticks = get_cpu_cycle() - ticks;
print_tick_count(ticks);

return EXIT_SUCCESS;
}

```

즉, 지역 변수는 이제 레지스터에 할당될 수 있는 기회를 얻을 수 있지만, 전역 변수는 여전히 메모리를 접근해야 한다. 당연히 실행 시간이 차이가 날 수밖에 없다. 경우에 따라 거의 1,000배에서 10,000배 이상의 실행 시간 차이가 발생할 수도 있다.

하지만, 아직도 위의 코드는 문제가 있다. 즉, 지역 변수("local")의 결과가 사용 되지 않기 때문에 컴파일러는 관련된 반복문 자체를 제거해 버린다. 따라서, 지역 변수가 사용 되도록 만들기 위해서는 간단하게 간단히 지역 변수의 값을 "printf()"를 추가해서 출력하도록 한다. 이 경우에는 지역 변수가 최적화에 크게 도움이 되지 않는다는 것을 확인할 수 있다.

```

...
print_tick_count(ticks);
printf("%d\n", local);
...

```

여기서 한가지 중요한 점은 전역 변수를 사용하는 이유가 성능 때문이라는 것은 설득력이 약하다는 것이다. 오히려 전역 변수 사용으로 인해서 코드에 버그가 발생할 가능성이 높아지고, 수정 및 이해가 어려운 코드가 만들어지는 것은 경험적으로 이미 사실이기 때문이다. 또한, 적절하게 지역 변수를 활용할 수 있는 방법을 적용한다면, 전역 변수를 사용해서 얻고 싶은 성능상의 이익도 충분히 거둘 수 있다는 점이다. 일반적으로 조금이라도 더 오버헤드(메모리에서 데이터를 읽고 저장하는 것은 레지스터를 사용하는 것 보다 비싸다)를 추가했으면 했지, 전역 변수를 사용하는 것이 지역 변수를 사용하는 것 보다 월등히 좋은 성능을 내지는 못한다.

다만 한 가지 유의해야 할 것은 지역 변수를 쓰기 위해서 함수로 파라미터를 전달해야 하는 오버헤드가 있다는 말은 할 수도 있다. 또한, 상위 계층에서 만들어진 데이터를 하위 계층에서 받아보기 위해서는 함수의 호출 경로가 길어질 경우 문제가 크다고 말할 수도 있을 것이다. 하지만, 그것이 전역 변수를 사용해야 하는 이유는 아니다. 데이터는 사용 되는 곳에서 가깝게 정의 되어야 하며, 규모가 큰 데이터를 전달하기 위해서는 항상 포인터나 대체 방법을 이용하는 것이 정답이다. 즉, 함수로 전달되는 파라미터는 많

아야 하나 정도 추가될 수 있으며, 다양한 필드를 묶어도 마찬가지로 구조체의 포인터 형태로 전달하면 된다.

따라서, 함수의 파라미터 수를 줄여 준다면, 레지스터를 가능한 이용하게 될 것이다. 물론, 중간에 호출되는 함수들 사이에서 사용 되는 레지스터의 내용이 메모리에 보관되어야 할 필요는 있지만, 그렇다고 전역 변수를 사용해서 얻는 성능 향상과 크게 다르다고 보기는 어렵다. 1GHz이상으로 동작하는 컴퓨터의 경우 몇 나노(Nano)초에 이루어질 것이 분명하다. 물론, 그 정도도 못마땅하다면 전역 변수를 사용해야 하겠지만, 그런 응용 프로그램이나 펌웨어(Firmware)는 정말 드물게 존재한다는 점에서 큰 이유는 안될 것이다. 오히려 그런 경우라면 하드웨어를 이용해서 데이터를 전달하는 것이 더 좋은 방법일지도 모른다.

지역 변수를 사용하게 되면 결과에 영향을 주지 않는 부분의 코드는 최적화 시 제거되지만, 전역 변수의 경우에는 제거되지 않는다. 즉, 컴파일러가 사용자의 의도를 모르기 때문에 최적화를 하지 못하고 그대로 코드를 남겨둔다. 누군가 변경된 값을 이용할 가능성이 충분히 있기 때문에 제거 하지 않고 최소한의 최적화만 한 후 멈춘다. 물론, 이 때도 지역 변수가 결과에 영향을 주게 되면 전역 변수를 사용하는 것과는 큰 차이를 보이지 않을 수도 있다. 전역 변수를 위해서 불러오고 저장하는데 필요한 몇 번의 연산 정도만 줄여 줄 뿐이다.

추가적으로 나중에 볼 “Volatile” 변수를 사용할 경우는 오히려 전역 변수를 사용하는 것보다 더 못한 수준의 최적화를 성능을 보인다. 특히, 반복문이 “Volatile” 변수의 사용과 관련이 있을 경우나, 관련이 없더라도 코드의 상하에 위치하는 경우에는 컴파일러가 최적화 자체를 못하게 만든다. 따라서, 전역 변수를 사용하는 것 보다 못한 결과를 낳을 수 있다. 따라서, 변수를 사용할 때의 순서는 지역 변수, 전역 변수, “Volatile” 변수의 순으로 최적화에 도움이 된다는 것을 기억하는 것이 좋을 것이다.

결론적으로 이야기 하면, 될 수 있으면 전역 변수를 쓰지 않는 것이 최적화에 도움이 된다. 어쩔 수 없이 써야 한다면, 반복문 내에서는 사용을 자제해야 하는 것이 좋다. 최적화를 했다고 생각된다면 반드시 측정해서 결과를 확인해야 하며, 최적화 옵션으로 생성된 어셈블리(Assembly) 코드를 직접 비교해서 정말 최적화가 되었는지 눈으로 봐야 한다. 컴파일러들은 컴파일 옵션으로 어셈블리 코드를 생성하도록 만들 수 있는데, 최적화 전과 후의 어셈블리 코드를 비교하는 것으로 원하는 목적을 이룰 수 있을 것이다

[주의]

위에서 제공하는 프로그램은 실행시 문제를 일으킬 수 있다. 즉, “Windows”는 기본으로 제공하는 프로그램의 스택(Stack)의 크기를 1MB정도만 할당한다. 따라서, 그보다 더 큰 크기를 가지는 지역변수(여기서는 “local”)에 대한 접근은 잘못된 주소 접근을 발생시켜 프로그램의 실행이 중단될 수 있다. 이때는 사용하는 개발자 환경에서 스택 크기를 따로 지정해 주어야 할 것이다. 예를 들어, “Windows”에서는 프로젝트의 설정에서 설정을 살펴보기 바란다. 한다. “Mac OS X”에서는 기본 스택 크기가 8MB로 되어 있어서 문제가 발생하지 않는다. 하지만, 그보다 큰 메모리를 스택으로 사용해야 한다면 마찬가지로 스택 설정을 변경할 수 있는 방법을 찾기 바란다.

[“volatile” 변수와 최적화]

“volatile”이라고 선언된 변수가 사용 되는 범위에서는 컴파일러는 제대로 최적화를 하지 못한다. 완전히 안하는 것이 아니라, 제한된 수준에서만 최적화를 시킨다. “volatile”은 언제든 코드의 다른 곳에서 값의 변경이 있을 가능성이 있기에, 변수의 값을 사용하는 곳을 컴파일러가 알 수 없다. 즉, 명령문의 순서를 바꾸거나 축약하는 형태의 최적화를 수행하지 못한다. 따라서, 최적화가 정말 필요한 곳에서는 될 수 있으면 “volatile”로 선언된 변수를 사용하지 않는 것이 좋다.

```
#define MAX_ARRAY_LENGTH 1000000
```

```
int main(void) {
```

```

puts("Volatile Variable Example 01");
unsigned int ticks;
unsigned int array[MAX_ARRAY_LENGTH];
volatile unsigned int result = 0; /* volatile로 선언할 경우 최적화에 영향을 준다. */
unsigned int result = 0;

memset(array, 0, sizeof(array));
ticks = get_cpu_cycle();
for (unsigned int i = 1; i < MAX_ARRAY_LENGTH; i++) {
    array[i] = i + i;
    array[i] *= i;
    array[i] -= i;
    array[i] /= i;
    result += array[i];
}
ticks = get_cpu_cycle() - ticks;
print_tick_count(ticks);

return EXIT_SUCCESS;
}

```

위의 코드에서는 인위적으로 “result”라는 변수를 “volatile”로 선언해 주었다. 이 경우 컴파일러는 반복문에 대한 최적화를 제대로 하지 못하게 된다. 반드시 사용해야 하는 경우가 아니라면, 다른 변수에 잠시 저장 했다가 나중에 다시 값을 복구하는 방식을 사용하는 것이 성능 개선에는 도움이 될 것이다. 특히, 반복문과 같이 실행이 잦은 곳에서는 “volatile”로 선언된 변수의 사용은 가능한 자제하는 것이 좋다.

```

...
memset(array, 0, sizeof(array));
volatile unsigned int temp = result; /* 선언된 곳과 사용되는 곳이 차이가 크다. */

ticks = get_cpu_cycle();
for (unsigned int i = 1; i < MAX_ARRAY_LENGTH; i++) {
    array[i] = i + i;
    array[i] *= i;
    array[i] -= i;
    array[i] /= i;
    result += array[i];
}
//volatile unsigned int temp = result; /* 이곳에 선언해서 사용하는 것이 유리하다. */
ticks = get_cpu_cycle() - ticks;

printf("%u\n", result );
printf("%u\n", temp);
print_tick_count(ticks);
...

```

“volatile”로 변수를 선언하는 위치도 중요하다. 반복문을 사이에 두고 정의와 사용이 있게되면, 중간에 들어가 있는 코드들은 최적화가 되지 않는다. 따라서, “volatile”로 선언된 변수를 사용하기 위해서는 접근하는 연산과 저장하는 연산의 위치를 적절한 순간까지 될 수 있으면 미루는 것이 좋은 선택이 될 수 있을 것이다. “volatile”을 사용할 때도 전역 변수와 마찬가지로 최적화를 위해서 직접 측정 값을 확인해 봐야한다. 당연히 생성되는 어셈블리 코드도 함께 검토해야 할 것이다.

[루프 풀어(Loop Unrolling) 해치기]

C언어는 문법에 의한 제약에서 비교적 자유롭다. 하지만, 잘못 사용하면 그 책임은 전적으로 개발자의 책임이며, 최적화도 그 중에 일부라고 볼 수 있다. 컴파일러의 경우에도 언어 표준에 대해서만 일관된 행동을 보장해 줄 뿐이다. 따라서, 정의되지 않은 행동이 컴파일러들마다 다를 수 있다.

코딩은 최적화 보다는 구조화를 먼저 해야한다는 점이 중요하지만, 어쨌든 최적화는 항상 사람의 손을 거치는 것이 컴파일러에 의존하는 것보다 효과적인 경우가 많다. 최적화의 기본 원칙은 "최적화 할 부분을 먼저 찾으라"이다. 즉, 최적화 노력의 대부분은 효과적이지 않다는 것이며, 최적화를 하기 위해서는 먼저 무엇을 최적화 할 것인지를 측정을 통해서 알아내야 한다는 것이다. 다시 한번 이야기 하지만, 최적화 전에 구조화를 먼저 해야 한다. "구조화된 코드는 최적화가 쉽지만, 최적화 되었다고 말하는 코드들은 구조화 하기 어렵다". 당연히 최적화 과정에서 발생하는 변경에 대해서 유연하지 못한 구조는 성능의 개선도 크지 않으며, 고치는 비용에 비해서 버그 발생 가능성이나 관리에 대한 요구만 더 커질 뿐이다.

C언어로 작성된 코드의 대부분은 루프(Loop)를 돌면서 시간을 소비한다(다른 언어로 작성된 코드들도 대부분 마찬가지지만). 즉, "for()", "while()", "do ~ while()" 등의 실행문이 대부분의 시간을 소모한다. 따라서, 루프를 최적화하는 것이 프로그램의 성능에 큰 영향을 줄 수 있다. 루프는 실행하기하기 위해서는 조건을 만족시키는지 항상 검사해야 하며, 이는 일반적인 할당(Assign)문 보다 더 오랜 CPU 시간(CPU Cycle)을 요구한다. 따라서, 최대한 루프를 적게 실행하는 것이 성능 향상에도 더 좋다.

```
for( int i = 0; i < MAX_SIZE; i++ ) {
    array[ i ] = do_something();
}
```

위의 코드는 루프를 돌면서 배열(array[])에 "do_something()" 함수의 호출 결과를 "MAX_SIZE" 번 넣어주는 일을 한다. 첫 번째 문제는 루프 내에서 함수의 호출이 반복적으로 발생하고 있다는 점이다. 함수를 호출하기 위해서는 함수 호출 준비 과정이 필요하며, 호출에 관련된 파라미터의 스택이나 레지스터를 이용한 전달과, 복귀값의 저장과 가져오기 등의 일이 발생해야 한다. 따라서, 함수가 비교적 간단한 일만 수행한다면 함수 자체를 "inline"화 시키는 것이 유리하다. 즉, 함수 호출에 대한 오버헤드가 없기 때문이다. 하지만, 함수 호출 회수가 많아지는 경우에는 컴파일러가 자동적으로 인라인 함수를 일반 함수 호출로 만들 가능성도 있다.

```
static inline do_something( void ) {
    ...
}

for( int i = 0; i < MAX_SIZE; i = i + 2 ) { /* 두번의 수행이 있었기에, 2씩 증가 */
    array[ i ] = do_something();
    array[ i + 1 ] = do_something(); /* 반복 회수를 줄이기 위해서 한번 더 호출 */
}
```

[주의]

만약, MAX_SIZE가 증가하는 수의 배수가 되지 않는다면, 종료 조건에 따라 값을 따져주어야 할 수도 있다. 따라서, 될 수 있으면 증가하는 수의 배수가 루프의 종료 조건에 맞도록 해주는 것이 좋다. 예를 들어, 100번 수행해야 하는 반복문인 경우 "3"으로 증가시킬 경우에는 마지막 부분에서 루프를 빠져나와 추가적으로 100번을 채워주는 처리를 해야한다.

반복 회수를 줄이기 위해서는 반복해서 실행해야 하는 명령문을 늘려줄 수 있다. 그리고, 그렇게 늘어난 실행해야 할 명령문을 병렬적으로 실행할 수 있는 가능성도 있다. 예를 들어, 영향을 받는 자료구조가 완

전히 분리되어 있다면, 각각을 분산(Distributed)해서 처리할 수도 있을 것이다. 반복의 회수를 줄이기 위해서 복잡한 연산을 해서는 안된다. 얻는 효과보다 더 많은 오버헤드가 따른다면 그렇게 만들 이유는 없다.

반복 회수를 줄이기 위해서 빨리 조건을 빠져나올 수 있게 만들거나, 혹은 증가보다는 감소하는 방향으로 반복문을 제어하는 변수를 사용할 수도 있다. 즉, "for(int i = 100; i >= 0; i--){}"와 같이 코딩할 수도 있다. 어떤 것이 코드를 더 간소화(읽기 쉽게) 할 수 있는지를 판단해서 적절한 방법을 사용하면 된다. 물론, 이렇게 만들었을 때는 일반적인 루프와는 다르기에 루프 내부에서 코딩에 주의해야 할 필요는 있다. 즉, 코드의 가독성을 위해서는 권장하지 않는 방법이다. 가독성이 중요하다면 일반적으로 많이 쓰는 형식으로 표현해야 할 것이다.

[배열 초기화에 대한 최적화]

여러 개의 배열이 같이 사용될 경우 같은 크기를 가질 가능성이 높다. 이런 경우에 배열을 초기화 해야한다면, 한번에 같이 하는 것이 좋다. 배열을 특정 값으로 정해야하는 경우는 흔히 있으며, 이미 있는 배열의 내용을 복사해야 하는 경우도 있다. 이때도 가능한 빠른 함수를 이용하는 것이 효과적일 것이다.

```
#define LENGTH 1000000

int main(void) {
    puts("MemorySet Example 01"); /* prints MemorySet Example 01 */
    unsigned int array_X[LENGTH];
    unsigned int array_Y[LENGTH];
    unsigned int ticks;

    /* 배열 각각을 나누어서 초기화 */
    ticks = get_cpu_cycle();
    for (unsigned int i = 0; i < LENGTH; i++) {
        array_X[i] = i;
    }
    for (unsigned int i = 0; i < LENGTH; i++) {
        array_Y[i] = i;
    }
    ticks = get_cpu_cycle() - ticks;
    print_tick_count(ticks);

    /* 배열을 함께 초기화 */
    ticks = get_cpu_cycle();
    for (unsigned int i = 0; i < LENGTH; i++) {
        array_X[i] = i;
        array_Y[i] = i;
    }
    ticks = get_cpu_cycle() - ticks;
    print_tick_count(ticks);

    return EXIT_SUCCESS;
}
```

위의 코드에서는 각각을 나누어서 초기화하는 것과 함께 초기화하는 것을 비교했다. 당연히 함께 초기화하는 것이 반복 회수를 줄여주기에 더 좋은 결과를 보여줄 것이다. 하지만, 이것도 최적화를 했을 때는

차이가 미미해 질 수 있다. 이것도 마찬가지로 초기화에 어떤 일을 해야하는가에 따라 최적화 옵션을 적용한 결과가 달라질 수 있다는 점을 주의해야 한다.

아래의 코드는 배열의 초기화 하는 다양한 방법들을 보여 준다. 각각의 경우는 배열을 나누어서 초기화하기, 배열을 한번에 함께 초기화하기, 배열을 “memset()”으로 특정값으로 초기화 하기, 배열을 다른 배열에 복제하기, 배열을 “memcpy()”를 사용해서 다른 배열로 복제하기를 구현했다.

```
#include <string.h>
#define LENGTH 1000000

int main(void) {
    puts("MemorySet Example 01");
    unsigned int array_X[LENGTH];
    unsigned int array_Y[LENGTH];
    unsigned int ticks;

    /* 배열을 각각 나누어서 초기화 */
    printf("CPU ticks for each array initialization!!!\n");
    ticks = get_cpu_cycle();
    for (unsigned int i = 0; i < LENGTH; i++) {
        array_X[i] = 0;
    }
    for (unsigned int i = 0; i < LENGTH; i++) {
        array_Y[i] = 0;
    }
    ticks = get_cpu_cycle() - ticks;
    print_tick_count(ticks);

    /* 배열을 한번에 초기화 */
    printf("CPU ticks for simultaneous array initialization!!!\n");
    ticks = get_cpu_cycle();
    for (unsigned int i = 0; i < LENGTH; i++) {
        array_X[i] = 0;
        array_Y[i] = 0;
    }
    ticks = get_cpu_cycle() - ticks;
    print_tick_count(ticks);

    /* 배열을 “memset()”으로 초기화 */
    printf("CPU ticks for array initialization with memset()!!!\n");
    ticks = get_cpu_cycle();
    memset(array_X, 0, LENGTH * sizeof(unsigned int));
    memset(array_Y, 0, LENGTH * sizeof(unsigned int));
    ticks = get_cpu_cycle() - ticks;
    print_tick_count(ticks);

    /* 배열을 다른 배열의 값으로 초기화 */
    printf("CPU ticks for array copy with assignment!!!\n");
    ticks = get_cpu_cycle();
    for (unsigned int i = 0; i < LENGTH; i++) {
        array_X[i] = array_Y[i];
    }
}
```

```

        array_Y[i] = array_X[i];
    }
    ticks = get_cpu_cycle() - ticks;
    print_tick_count(ticks);

/* 배열을 “memcpy()”로 다른 배열을 복사해서 초기화 */
printf("CPU ticks for array copy with memcpy()!!!\n");
ticks = get_cpu_cycle();
memcpy(array_X, array_Y, LENGTH * sizeof(unsigned int));
memcpy(array_Y, array_X, LENGTH * sizeof(unsigned int));
ticks = get_cpu_cycle() - ticks;
print_tick_count(ticks);

return EXIT_SUCCESS;
}

```

대체로 배열을 초기화 할 때나 다른 배열을 복제할 경우에는 “memset()”이나 “memcpy()”로 하는 것이 성능이 더 좋다. 물론 이것도 초기화하는 값과 컴파일러의 최적화 옵션에 따라 차이를 보이지 않을 수 있다. CPU의 성능이 충분히 좋다면, 컴파일러의 옵션(Option) 만으로도 충분히 최적화를 할 수 있지만, 그렇지 않다면 배열의 설정에 메모리 설정 관련 함수들을 사용하는 것도 좋은 방법이다.

주의할 점은 “memset()”이나 “memcpy()”는 바이트(Byte)단위로 동작한다는 점이다. 따라서, 복제할 배열의 크기를 줄 때 주의해야 하며, 복제할 타입이 바이트 이상의 크기일 경우에는 타입의 크기와 복제할 요소의 숫자를 곱해서 정확한 크기를 구해야 한다. 특히, “memset()”의 경우에는 “unsigned char” 형으로 된 자료를 복제하기에 주로 “0”이나 “1”을 설정하는데 유리하다.

[배열 접근에 대한 최적화]

배열은 접근을 위해서 항상 반복이나 인덱스의 계산을 필요로 한다. 즉, 루프와 함께 사용 되는 경우가 많다. 따라서, 최적화의 대상으로 생각할 만한 충분한 이유가 있는 것이다. 배열은 주로 자료의 저장을 위해서 사용하며, 2차원 배열로 사용할 경우에는 저장된 원소의 크기 및 행과 열의 인덱스를 계산해서 배열의 원소에 접근하게 된다.

```

#define HEIGHT 100
#define WIDTH 1000

static unsigned int array[HEIGHT][WIDTH];

int main(void) {
    puts("Array Optimization 01");
    unsigned int ticks;

    /* 행->열 순으로 접근 */
    ticks = get_cpu_cycle();
    for (unsigned int i = 0; i < HEIGHT; i++) {
        for (unsigned int j = 0; j < WIDTH; j++) {
            array[i][j] = i * j;
        }
    }
    ticks = get_cpu_cycle() - ticks;
    print_tick_count(ticks);
}

```

```

/* 열->행 순으로 접근 */
ticks = get_cpu_cycle();
for (unsigned int i = 0; i < WIDTH; i++) {
    for (unsigned int j = 0; j < HEIGHT; j++) {
        array[j][i] = i * j;
    }
}
ticks = get_cpu_cycle() - ticks;
print_tick_count(ticks);

return EXIT_SUCCESS;
}

```

위의 코드는 배열을 중첩된 루프를 통해서 배열의 개별 원소에 접근하는 것을 보여주고 있다. 이때, 행과 열을 계산하는데 있어서 어떤 것이 더 효과적인지를 알기 위해서 간단히 실험해 본 것이다. 대부분의 경우 “행을 먼저 접근하고, 열을 접근하는 것”이 더 좋은 성능을 보인다. 배열은 특정 행의 원소를 찾기 위해서는 곱셈 계산이 추가되기 때문이다. 열의 크기를 줄이더라도 비슷한 결과를 보인다. 따라서, 2차원 배열을 사용해서 각각의 원소에 연산을 해줄 경우에는 “행->열”방향으로 움직이는 것이 성능에 도움을 줄 것이다.

다음의 코드는 첫 번째는 전혀 최적화를 하지 않은 상태에서 배열에 대한 인덱스 계산과 루프를 실행 했으며, 두 번째는 반복의 회수를 줄여준 것이고, 세 번째 것은 배열에 대한 인덱스 계산과 반복의 회수를 둘 다 줄인 것이다. 실행 시마다 결과가 다르게 나올 수 있지만, 대체로 뒤로 갈수록 더 좋은 결과를 보여 준다.

```

#define WIDTH 1000
static unsigned int result[WIDTH][WIDTH];
static unsigned int inputX[WIDTH];
static unsigned int inputY[WIDTH];

int main(void) {
    puts("Array Indexing Optimization");
    uint64_t ticks = 0;

    /* 배열을 초기화 시킴 */
    for (int i = 0; i < WIDTH; i++) {
        inputX[i] = i;
    }
    for (int i = 0; i < WIDTH; i++) {
        inputY[i] = i;
    }

    /* (1) 최적화를 전혀 안함 */
    ticks = get_cpu_cycle();
    for (unsigned int i = 0; i < WIDTH; i++) {
        for (unsigned int j = 0; j < WIDTH; j++) {
            result[i][j] = inputX[i] * inputY[j];
        }
    }
    ticks = get_cpu_cycle() - ticks;
    print_tick_count(ticks);
}

```

```

/* (2) 반복 회수를 줄임 */
ticks = get_cpu_cycle();
for (unsigned int i = 0; i < WIDTH; i++) {
    for (unsigned int j = 0; j < WIDTH; j = j + 2) {
        result[i][j] = inputX[i] * inputY[j];
        result[i][j] = inputX[i] * inputY[j + 1];
    }
}
ticks = get_cpu_cycle() - ticks;
print_tick_count(ticks);

/* (3) 배열 인덱스 계산을 줄임 */
ticks = get_cpu_cycle();
for (unsigned int i = 0; i < WIDTH; i++) {
    unsigned int xtemp = inputX[i];
    for (unsigned int j = 0; j < WIDTH; j = j + 2) {
        result[i][j] = xtemp * inputY[j];
        result[i][j] = xtemp * inputY[j + 1];
    }
}
ticks = get_cpu_cycle() - ticks;
print_tick_count(ticks);

return EXIT_SUCCESS;
}

```

결과는 첫 번째와 두 번째는 분명하게 차이가 나지만, 두 번째와 세 번째는 큰 차이를 보이지 않는다. 특히, 반복의 회수가 늘어나면 더 극적인 차이를 보여 준다는 것을 알 수 있을 것이다.

[“switch()”의 최적화]

“switch()”의 분기가 많아지면, 분기문을 사용하는 것보다 배열을 이용해서 처리하는 방법을 사용할 수 있다. 즉, “switch()”문의 각 케이스(case)를 인덱스로 활용해서 배열에 지정된 함수 포인터를 실행하는 방법이다. 이 방법은 “switch()”문의 확장이 많아질 경우에 유용하며, 좀 더 개선하면 코드의 유지보수(확장성) 측면에서도 편의를 제공해 줄 수 있다.

아래의 코드는 제품의 형에 따라 특정 일을 해주어야 할 경우를 가정한다. 여기서는 단순히 아무 일도 하지 않는 함수를 호출하는 것으로 대신 하도록 했다.

```

typedef enum PRODUCT {
    PRODUCT_A = 0, PRODUCT_B = 1, PRODUCT_C = 2, PRODUCT_D = 3,
} PRODUCT_TYPE;

typedef struct SWITCH {
    void (*func)(void);
} SWITCH_TABLE;

void do_nothing(void) {
    return;
}

```

```

#define TIMES 1000
int main(void) {
    puts("Switch Optimization Example 01");
    uint64_t ticks = 0;
    uint64_t result = 0;
    unsigned int product_type;
    SWITCH_TABLE table[ 4 ] = {{do_nothing}, {do_nothing}, {do_nothing},
{do_nothing}};

    /* 반복내에서 분기문의 실행 */
    for (unsigned int i = 0; i < TIMES; i++) {
        product_type = i % 4;
        ticks = get_cpu_cycle();
        switch (product_type) {
            case PRODUCT_A:
                do_nothing();
                result += get_cpu_cycle() - ticks;
                break;
            case PRODUCT_B:
                do_nothing();
                result += get_cpu_cycle() - ticks;
                break;
            case PRODUCT_C:
                do_nothing();
                result += get_cpu_cycle() - ticks;
                break;
            case PRODUCT_D:
                do_nothing();
                result += get_cpu_cycle() - ticks;
                break;
            default:
                break;
        }
    }
    print_tick_count(result);

    /* 반본 내에서 테이블을 이용한 분기 */
    result = 0;
    for (unsigned int i = 0; i < TIMES; i++) {
        product_type = i % 4;
        ticks = get_cpu_cycle();
        table[ product_type ].func();
        result += get_cpu_cycle() - ticks;
    }
    print_tick_count(result);

    return EXIT_SUCCESS;
}

```

첫 번째는 “switch()”를 이용해서 분기하는 것을 각각의 경우에 대해서 반복적으로 실행 했으며, 두 번째는 단순히 배열에 설정된 함수를 인덱스를 증가 시키며 실행 했을 뿐이다. 각각의 경우를 비교하면, 대체

로 첫 번째 경우가 더 많은 실행 시간이 걸린다는 것을 확인할 수 있을 것이다. 매번 같은 결과는 나오지 않지만, 더 많은 경우 첫 번째가 더 오래 시간이 걸린다.

[반복문 내에서 함수 사용하지 않기]

반복문 내에서 함수의 사용은 자제해야 한다. 만약, 그렇게 꼭 사용해야 할 경우라면, 차라리 반복문 전체를 함수 내부로 옮기는 것이 좋다. 물론, 그렇게 했을 때 함수 내부에서 필요한 자료 구조에 대한 부분을 포인터와 같은 형태로 함수 호출시 파라미터로 전달해 주어야 한다.

```
for( int i = 0; i < MAX_NUMBER_OF_MEMBER; i++ ) {
    result[ i ] = calculate_height( member, i );
}
```

이 경우에는 "for()" 루프를 돌면서 "calculate_height()" 함수를 최대 "MAX_NUMBER_OF_MEMBER" 만큼 호출해야 한다. 따라서, 함수 호출 오버헤드가 최대 "MAX_NUMBER_OF_MEMBER" 만큼 곱으로 늘어나게 된다.

```
calculate( member[], result[], MAX_NUMBER_OF_MEMBER );
...
void calculate( const unsigned int member[], const unsigned int result[], unsigned int size ){
    for ( int i = 0; i < size; i++ ) {
        ...
    }
}
```

앞에서 "for()" 루프를 돌면서 호출했던 함수 내부로 관련된 루프를 이동시켜서 구현한 것이다. 이 경우에는 함수 호출 오버헤드는 항상 한번의 호출에 불과하며, 다만 전달되는 파라미터가 변경 되었을 뿐이다. 그리고, 파라미터의 수가 한 개 더 추가되었다. 당연히 함수 호출에 대한 오버헤드의 감소로 인해서 성능은 그 만큼 개선될 것이다.

[반복문 내에서 조건문 줄이기]

조건문은 오버헤드가 크다. 따라서, 조건문의 수를 줄이는 것도 중요한 성능 향상 방법이다. 조건문이 반복문 내부에 있을 경우에는 매번 반복마다 조건문은 실행되어야 하기 때문에, 가능한 그것을 줄여주면 성능상의 이득을 볼 수 있을 가능성이 높다. “가능성”이라는 말에는 일반적으로 그럴 것이라는 예상이 항상 옳은 것은 아니라는 전제를 가지고 있다.

```
#define MAX_TIMES 1000000
enum {
    FIRST = 0, SECOND
};

void init(unsigned int array[], unsigned int size) {
    for (unsigned int i = 0; i < size; i++) {
        array[i] = i;
    }
}

int main(void) {
    puts("Conditional Expression Example 01");
    unsigned int flags = FIRST;
    unsigned int first[MAX_TIMES];
```

```

unsigned int second[MAX_TIMES];
unsigned int ticks;

init(first, MAX_TIMES);
init(second, MAX_TIMES);

/* for()에서 if()을 실행 */
ticks = get_cpu_cycle();
for (unsigned int i = 0; i < MAX_TIMES; i++) {
    if (flags == FIRST) {
        first[i] = i * i;
    } else if (flags == SECOND) {
        second[i] = i + i;
    }
}
ticks = get_cpu_cycle() - ticks;
print_tick_count(ticks);

/* if()문 내에서 for()문 실행 */
ticks = get_cpu_cycle();
if (flags == FIRST) {
    for (unsigned int i = 0; i < MAX_TIMES; i++) {
        first[i] = i * i;
    }
} else {
    for (unsigned int i = 0; i < MAX_TIMES; i++) {
        second[i] = i + i;
    }
}
ticks = get_cpu_cycle() - ticks;
print_tick_count(ticks);

return EXIT_SUCCESS;
}

```

위의 코드에서는 “반복문 -> 조건문”과 “조건문 -> 반복문”을 각각 바꾸어서 시간이 얼마나 걸리는지를 측정하고 있다. “반복문->조건문”的 경우에는 반복문이 “MAX_TIMES” 만큼 수행되고 각각의 경우에 대해서 조건문이 “MAX_TIMES”만큼 다시 수행된다. “조건문->반복문”的 경우에는 조건문이 1번 수행되고, 각각의 경우에 대해서 “MAX_TIMES”만큼의 반복문이 실행된다. 당연히 전자가 시간이 더 많이 걸릴 것이라는 예상을 할 수 있다.

하지만, 실제로 이 코드를 실행하면 별로 이득을 보지 못할 수도 있다. 간단한 예로 만든 코드 이기에 실제로 조건문이나 반복문 내에서 실행해야 하는 코드가 무엇인지에 따라 달라질 수 있으며, 최적화 수준을 높였을 때 큰 성능 향상을 기대하기 어려운 경우도 있다. 따라서, 이 방법도 어느 것이 더 읽기 편한지를 따져서 선택해야 할 것이다. 성능과 코드의 가독성은 대부분의 경우 반비례 관계를 가지기 때문이다.

[참고]

ARM CPU를 사용하는 경우에는 최적화를 위해서 ARM CPU의 특성에 맞출 필요가 있다. 아래는 ARM CPU에서 C코드의 최적화를 잘 다루고 있다고 생각되는 웹 주소다. 이곳에 있는 것들과 ARM에서 제공하는 각종 메뉴얼을 이용해서 C코드의 최적화를 위한 방법을 스스로 익혀나갈 수 있을 것이다

<http://www.codeproject.com/Articles/6154/Writing-Efficient-C-and-C-Code-Optimization>

맺음말

어떤 프로그램을 만들 것 인가는 개발자의 의지에 달려있다. 하지만, 그 의지가 자신의 인생과 타인에게 영향을 줄 수 있다는 생각이 든다면, “좋은 코드를 만들려는 노력”은 멈추지 않아야 할 것이다. 그것이 최소한 나와 내가 속한 사회 및 조직에 대한 감사의 마음을 보내는 것 이기 때문이다.

프로그램은 인류가 만든 것 중에서도 가장 복잡한 것에 속한다. 좋은 코딩이란 복잡함을 다룰 수 있는 수준에서 유지하는 방법을 말하며, 복잡한 문제를 간결하게 표현하는 것은 오랜 경험과 지식에서 나온다. 하지만, 그런 경험들이 이미 축적되어 있다면 이야기는 달라진다. 즉, 익히는 사람의 자세에 따라 결과가 달라질 수 있다는 것이다. 없는 것을 만들라는 것이 아니라, 이미 있는 것들을 잘 활용 하라는 것이다.

프로란 결국 누군가의 기대를 만족 시키며 그 댓가를 얻는 사람들이다. 프로의 직업정신이란 “정직”과 “전문성”이며, 어떤 상황에서도 만족스러운 결과를 만들기 위해서 자신의 모든 노력을 다 하는 사람이다. 소프트웨어 개발이란 사용자의 만족 추구와 동시에 내부 이해 관계자의 기대를 충족시키는 활동이다. 일정, 비용, 기능의 만족만이 중요한 것이 아니라, 성장을 위해서는 품질의 만족이 핵심으로 추구해야 할 목표다. 결국 내부적인 품질이 만족되지 못한 소프트웨어는 시장에서 사라지기 마련이다. 프로란 그런 기대를 충족시키기 위한 전문화 된 지식을 가지고 있으며, 이미 잘 알고 있는 것을 구현으로 옮기는 정직하고 투명한 활동을 하는 사람이다.

스스로 만족감을 얻기 위해서는 자신의 일에 대해서 자부심을 가져야 한다. 스스로 일의 결과에 대해서 “불안감”을 느낀다면, 그 자체가 일의 흥미를 떨어뜨리게 된다. 품질 확보란 결국 자신의 만족감을 높이기 위해서 “불확실성”을 제거하는 활동인 것이다. “작게 조금씩 만들고 꾸준히 반복적으로 검사하는 것”이 코드 품질의 기초를 만들어 주며, 부품 단위의 품질이 전체 제품의 품질과 이어지도록 한다. 단순히 빨리 만드는 것은 양으로 생산성을 취급하는 사람들이 하는 일이다. 소프트웨어 개발자는 품질로 말해야 한다. 자신의 코드에서 품질을 확인할 수 없다면, 스스로의 불안감만 키우게 되는 것이다.

소프트웨어 개발은 시간과의 싸움이다. 언제나 시간은 부족하다. 더 많이 하려고 하면 더 적게 하게되고, 더 빠르게 가려고 하면 더 늦어지게 되는 것이 소프트웨어 과제다. 두 걸음 걸어가기 위해서는 한 걸음 뒤로 물러 서기도 해야한다. 두려운 마음에 앞으로만 빨리 가려고 한다면, 다시 돌아가야 할 길만 더 길어질 뿐이다. 하지만, 현실은 무시할 수 없다. “일정은 생명이고, 품질은 자존심”이기에, 우리에게는 언제나 생존의 욕구를 앞세운다. 하지만, 그것만으로는 부족하다. 우리의 자존심도 지킬 수 있어야 한다. 그것이 보장되지 못한다면 우리 스스로 자신의 가치를 낮출 뿐이다.

개발자의 문화는 누군가가 만들어주는 것이 아니라 스스로 만들어 가야 한다. 좋은 문화가 없다고 남탓만 한다고 상황은 개선 되지는 않는다. “작지만 큰 첫 걸음”을 내딛기 위해서는, 누군가는 달나라로 가는 위험을 감수해야 하기 때문이다. 이제 그 작은 시작을 위한 첫 번째 책을 썼다. 누군가에게는 도움이 되겠지만, 누군가에게는 분란이 될 수도 있다. 하지만, 중요한 것은 이제 나의 이야기를 했다는 것이고, 다른 사람의 이야기를 들을 준비가 되었다는 것이다. 혼자만의 이야기 보다는 대화가 모든 변화의 핵심이기 때문이다.

- “완벽이란 더 빨 것 없는 상태”를 말한다”[생떼쥐페리].-