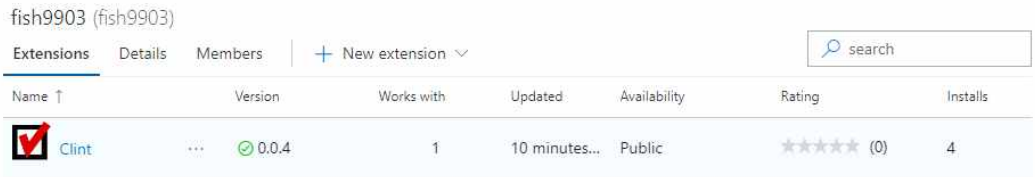
개별연구 보고서

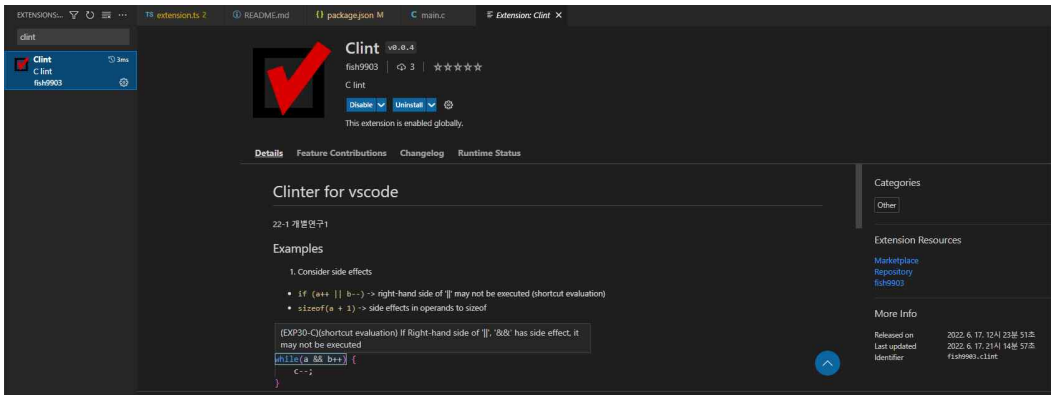
학생 개별 작성용

수행 학기	2022학년도 1학기			
교과목 정보	교과목명	CS 웹 IDE 기반 문제은행 연구	분반	-분반
교과담당교수	소속	컴퓨터공학	교수명	손 윤 식
학생정보	이름	조광호	학번	2018112072
	학과	컴퓨터공학	전화번호	010-8837-6702
	학년	3	이메일	fish9903@naver.com

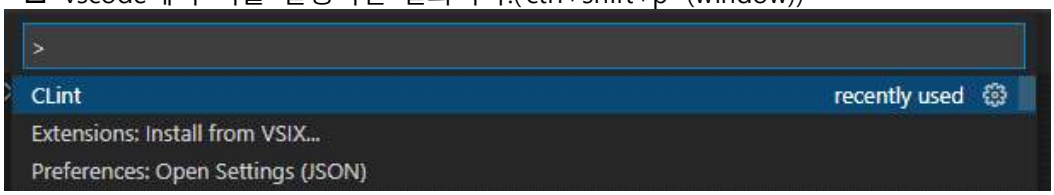
요약 보고서

작품명 (프로젝트명)	**CS 웹 IDE 기반 문제은행 연구(vscode C언어 코딩 패턴 검사/알림 extension)**
1. 연구·개발 동기/목적/ 필요성 및 개발 목표	□ 연구/개발의 목적은 사용자가 코드를 작성할 때 특정 언어에서 권장되는 혹은 사용자가 직접 지키고자 하는 코딩 스타일, 패턴을 지켰는지 검사함으로써, 올바른 코딩 습관을 지닐 수 있도록 함이다. □ 특정 언어에서 권장하는 코딩 스타일을 따르지 않고 코드를 작성하면 오류가 발생할 수 있고, 오류가 발생하지 않아도 프로그램에 좋지 않은 영향을 줄 수 있다. 따라서 적절한 코딩 스타일을 따르는 것이 필요하다. □ 샤스트라 웹 IDE(vscode)에서 작성하는 C 코드를 분석하여 특정 코딩 패턴을 지켰는지 위배했는지 검사하는 vscode extension(plugin)을 만드는 것이 목표이다.
2. 최종 결과물 소개	□ 구현한 extension을 visual studio marketplace로 배포를 하였다. 

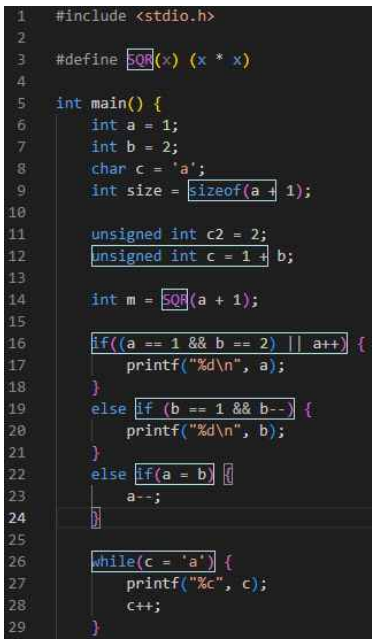
□ visual studio code에서 이 extension을 받아서 사용할 수 있다.



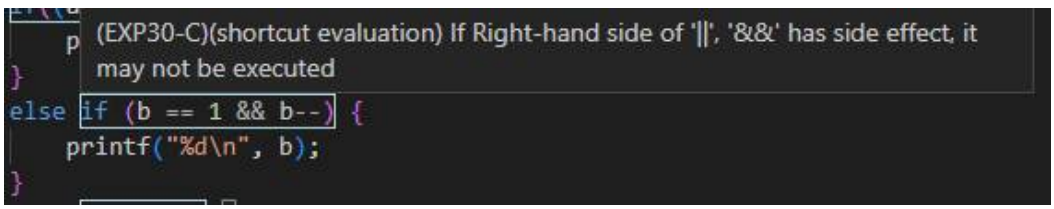
□ vscode에서 이를 실행시킨 결과이다.('ctrl+shift+p' (window))

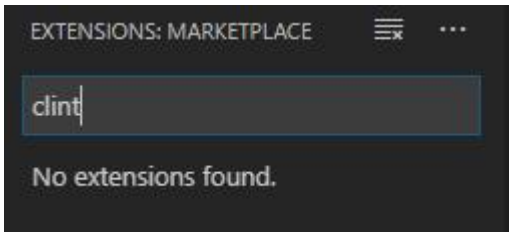


구현할 때 작성해둔 코딩 패턴에 해당하는 코드 부분이 표시되고, 이 부분에 마우스를 올리면 설명이 다음과 같이 나온다.



```
#include <stdio.h>

#define SQR(x) (x * x)

int main() {
    int a = 1;
    int b = 2;
    char c = 'a';
    int size = sizeof(a + 1);

    unsigned int c2 = 2;
    unsigned int c = 1 + b;

    int m = SQR(a + 1);

    if((a == 1 && b == 2) || a++) {
        printf("%d\n", a);
    }
    else if (b == 1 && b--) {
        printf("%d\n", b);
    }
    else if(a = b) {
        a--;
    }

    while(c = 'a') {
        printf("%c", c);
        c++;
    }
```



3. 프로젝트 추진 내용	□ 분석 대상 언어는 C언어로 하였고 코딩 패턴은 CERT Secure Coding Standard를 참고하여 적절한 코딩 패턴을 참고하였다. □ 최종적으로 샤스트라 웹 IDE에서 만든 extension을 테스트해보고자 했으나, 샤스트라 웹 IDE에서 생성한 프로젝트에서 이 extension을 찾을 수 없어 테스트해보지 못했다.  □ 현재 구현한 검사하는 코딩 패턴에 대한 설명은 아래의 '본론' 부분에 나와 있다.
4. 기대효과	□ 사용자는 vscode 환경에서 C 코드를 작성하고 해당 extension을 실행시켜봄으로써, 권장되는 코딩 패턴(cert secure coding standard 기반)을 참고하여 이 규칙을 지켰는지, 혹은 내가 작성하는 코드에서 잘못된 부분은 없지만 추후 잘못된 결과를 얻을 수 있는 경우가 어떤 경우인지 알 수 있다. 이를 통해, 언어의 문법상 오류는 없지만 추후 문제가 발생할 수 있는 부분을 사전에 찾고 방지하고, 사용자에게 이에 대한 지식을 전달할 수 있다. □ 또한 구현한 extension의 코드는 github에 올려놓고, 배포한 extesion의 설명에 github링크를 제공함으로써 다른 사람들이 추후 C 언어의 다른 권장되는 코딩 패턴을 지속해서 추가할 수도 있고, 사용자가 코드의 변경을 통해 자신이 검사하고 싶은 코딩 패턴을 정규 표현식 형태로 작성해 추가하면 사용자가 정의한 코딩 패턴 또한 검사하는 동작을 할 수 있게 바꿀 수 있다. 이러한 과정을 통해 extension이 더 확장 될 수 있을 것이다. github 링크: https://github.com/fish9903/Clint

1. 서론

코드를 작성할 때 사용한 언어 자체의 문법에서는 문제가 되지 않지만, 문제가 발생할 수 있는 코딩 패턴들이 존재한다. 그리고 이러한 문제는 구문 오류가 아니므로, 코드가 정상적으로 컴파일, 실행되어도 발생할 수 있다. 이런 오류를 논리 오류(Logical Error)라고 하는데 프로그램이 동작하긴 하지만, 코드를

작성한 사람이 의도한 결과대로 동작하지 않는 것을 의미한다. 논리 오류는 오류가 발생한 것을 알게 되어도, 어디에서 왜 발생했는지 찾기 어려운 경우가 많다. 논리 오류가 발생할 수 있는, 주의해야 할 코딩 패턴들은 CERT Secure Coding Standard에서 찾을 수 있다. (참고문헌 참고)

따라서 이러한 문제점을 미연에 방지하기 위해 코드를 작성할 때, 논리 오류가 발생할 수 있는 코딩 패턴들을 CERT Secure Coding Standard를 바탕으로 찾아, 코드를 작성하는 사람에게 알려주는 확장 프로그램(vscode extension)을 만들고자 한다.

2. 관련/배경 연구

2.1 'Side effect' in computer science

논리 오류(Logical Error)가 발생할 수 있는 상황을 찾아볼 때, 'side effect'라는 단어를 볼 수 있었다. 보통 side effect라는 단어는 '부작용'으로 해석되는데, computer science에서는 '부작용' 그대로 해석하는 것보다는, 조금 다르게 해석하는 것이 좋다.

Computer science에서 side effect란 실행 중에 어떤 객체에 접근해서 변화가 일어나는 행위를 의미한다. 예를 들어 'a = b + 1;' 표현식은 1개의 side effect가 있다. a의 값이 변경되었기 때문이다.
'a = b++;' 표현식은 2개의 side effect가 있다. b 값이 변경되고 이후, a 값도 변경되기 때문이다.

Side effect 자체는 문제가 없다. 하지만 코드를 side effect를 고려하지 않고 작성할 경우, 논리 오류가 발생할 수 있다. 예를 들어 if(a == 1 || b++) 조건문에서 문제가 되는 것은 '||' 조건문 이후 'b++'에 존재하는 side effect이다. C 언어에서는 if 조건문의 조건식을 판단할 때, 모든 조건을 확인하지 않아도 조건식의 결과를 알 수 있을 경우, 확인하지 않은 나머지 조건을 확인하지 않는 shortcut evaluation을 사용하기 때문에 '||'연산자 이후 'b++'가 실행되지 않을 수 있기 때문이다. 'b++'가 실행되는 것을 코드를 작성한 사람이 의도한 것인지, 실행되지 않는 것을 의도한 것인지 알 수 없지만, 둘 중 어느 경우에도 이러한 조건문에서는 논리 오류가 발생할 수 있어 바람직하지 않다.

2.2 Visual studio code extension(plug-in)

연구 주제가 웹 IDE에서 동작하는 visual studio code에서 작성하는 코드의 패턴을 분석하는 것이므로, visual studio code extension을 만드는 방법을 조사해보았다. Extension을 만드는 데 사용되는 패키지를 찾아서 사용하였고, 이를 사용해 기본으로 제공되는 extension을 테스트해 보았다.

npm install –g yo generator-code
yo code

```
TS extension.ts ×
src > TS extension.ts > activate
1  // The module 'vscode' contains the VS Code extensibility API
2  // Import the module and reference it with the alias vscode in your code below
3  import * as vscode from 'vscode';
4
5  // this method is called when your extension is activated
6  // your extension is activated the very first time the command is executed
7  export function activate(context: vscode.ExtensionContext) {
8
9      // Use the console to output diagnostic information (console.log) and errors (console.error)
10     // This line of code will only be executed once when your extension is activated
11     console.log('Congratulations, your extension "test" is now active!');
12
13     // The command has been defined in the package.json file
14     // Now provide the implementation of the command with registerCommand
15     // The commandId parameter must match the command field in package.json
16     let disposable = vscode.commands.registerCommand('test.helloWorld', () => {
17         // The code you place here will be executed every time your command is executed
18         // Display a message box to the user
19         vscode.window.showInformationMessage('Hello World from test!');
20     });
21
22     context.subscriptions.push(disposable);
23 }
24
25 // this method is called when your extension is deactivated
26 export function deactivate() {}
27
```

yo code로 test이름으로 extension을 만들었을 때, 기본으로 제공하는 extension 코드

```
>
Hello World                                recently used
```

extension 실행

```
C main.c ●
C main.c > main()
1  #include <stdio.h>
2
3  #define SQR(x) (x * x)
4
5  int main() {
6
7  }

Hello World from test!
```

extension 실행 모습. 'Hello World from test'라는 메시지가 나타나는 모습

이후 이것을 바탕으로 extension을 확장하기 위해 자료를 찾으면서 visual studio code extension reference와 microsoft에서 제공하는 visual studio code extension 예시 오픈소스를 찾아 분석하고 프로젝트에 적용하고자 하였다. (참고문헌 참고)

3. 본론

3.1 Side effect 예시 조사 및 선택

우선 Extension을 구현하기 전에 코드를 작성할 때 주의해야 할 side effect가 논리 오류를 발생시키는 코딩 패턴들과 side effect 문제가 아니라도 주의해야할 코딩 패턴들을 조사하였다. 이렇게 조사한 코딩 패턴들을 이후 extension에 적용하였다. CERT Secure Coding Standard의 C언어 문서를 참고하였다. 이 문서에서는 적절하지 않은 코딩 패턴과 이를 수정하는 방법을 제시하고, 해당 문제되는 코딩 패턴이 어떤 수준으로 위험한지 Priority(P)와 Level(L)을 사용해 표현하였다. Priority가 높고, Level이 낮으면 위험 순위가 높다는 뜻이다. Level은 1~3로 나눠지고, L3는 P1~4, L2는 P6~9, L1은 P12~P27로 나뉘어있다.

첫째, unsigned integer wrap 문제. 이는 CERT INT30-C 항목에 해당하는 문제로, unsigned integer에 값을 할당할 때 발생할 수 있는 문제이다. 여기서 wrap란 signed integer에서 발생하는 overflow와 비슷한 의미로, unsigned integer가 가질 수 있는 범위를 넘어가면 0부터 값이 시작하는 것이다. 이 항목을 선택한 이유는 대개 사람들은 signed integer의 overflow 현상은 잘 알고 사용할 때 주의한다. 하지만 unsigned integer는 overflow가 아닌 wrap현상이 일어나고, 이것에 대해 모르는 경우가 있기 때문에 선택하였다. CERT INT30-C 항목에 따르면 이러한 unsigned integer wrap는 +, -, *, ++, --, +=, -=, *=, <<=, << 연산자에서 발생할 수 있다고 한다. 따라서 코드를 작성할 때 unsigned integer에 값을 할당하는 연산이 있으면 찾아서 extension이 알려줄 수 있도록 할 것이다.
이 문제는 L2, P9에 해당하는 위험 정도를 가지고 있다.
(ex. unsigned int a = 9 * b; <- 검사 대상) -> 이후 바꾸면 좋을 추천 방법을 명시?

둘째, assignments in selection statements 문제. 이는 CERT EXP45-C 항목에 해당하는 문제로, if, while, for의 조건 부분에서 assignment 연산자(=)를 사용할 때 발생할 수 있는 문제이다. 예를 들어, while(a = 10) 과 같은 조건 부분은 a=10이 계산되고, while(10)꼴로 해석되어 별다른 탈출 구문이 없으면 무한 루프를 돌게 된다. 무한 루프를 도는 것을 코드 작성자가 의도했는지 알 수 없고, 설사 의도했더라도 이러한 코딩 패턴은 적절하지 않은 코딩 패턴이기 때문에 이를 선택하였다. (L2, P6에 해당하는 위험 정도) (ex. if(a = b) <- 검사 대상)

셋째, 각종 표현식에 존재하는 side effect를 고려하지 않았을 때 발생할 수 있는 문제점들. 여기에 해당하는 CERT 항목들은 다음과 같다.

- CERT EXP30-C: Do not depend on the order of evaluation for side effects
- CERT EXP44-C: Do not rely on side effects in operands to sizeof, ...
- CERT PRE31-C: Avoid side effects in arguments to unsafe macros

CERT EXP30-C 항목은 shortcut evaluation 때문에 특정 조건을 판단(실행)하지 않아 발생할 수 있는 문제이다. Shortcut evaluation은 조건문에서 조건을 판단할 때 두 개 이상의 조건이 존재하고, 이 모든 조건을 판단하지 않고 조건의 결과를 판단할 수 있으면 나머지 판단하지 않은 조건들을 판단하지 않고 건너뛰는 평가 방식이다. 이러한 평가 방식은 효율적이지만, 이를 고려하지 않고 코드를 작성하면 문제가 발생할 수 있다. 조건문에 side effect가 존재하는 경우가 그 예이다. 예를 들어 if(a==1 || b++)와 같은 조건식이 있다고 하자. 조건을 판단할 때 우선 a==1 조건을 먼저 판단할 것이다. 만약 a==1이 참이라고 판단이 되면 '||'연산자이기 때문에, 나머지 조건(b++)은 판단하지 않아도 된다. 이렇게 되면 'b++' 조건은 판단(실행)되지 않게 되고, b 값은 변하지 않는다. 하지만 a==1이 거짓으로 판단이 되면 'b++' 조건이 판단(실행)되고, b 값이 변한다. 이렇게 조건문에 side effect가 존재하는 코딩 패턴은 코드 작성자가 의도한 것일 수도 있지만, 이것 또한 적절한 코딩 패턴이 아니고, 이후 논리 오류가 발생할 수 있어 선택하였다. (L2, P9에 해당하는 위험 정도)

CERT EXP44-C 항목은 sizeof와 같은 연산자의 operand에 side effect가 존재할 때 발생할 수 있는 문제이다. 예를 들어 sizeof(a++);라는 연산은 'a++'가 실행되지 않고 a의 값이 변하지 않는다. 아마도 코드 작성자가 sizeof연산 이후, 'a++'를 통해 a의 값을 1 증가시키려는 의도를 가지고 이러한 코드를 작성했겠지만, 의도대로 실행되지 않는 논리 오류가 발생한 것이다. 'sizeof(a); a++;'로 쪼개서 작성해야 의도한 대로 동작할 것이다. (L3, P3에 해당하는 위험 정도)

CERT PRE31-C 항목은 macro에 side effect가 있을 때 발생할 수 있는 문제이다. 몇 가지 예시를 들면 #define ABS(x) ((x) < 0 ? -(x) : (x)) 라는 매크로를 정의하고, ABS(++a)를 해보자. 매크로의 x에 ++a가 들어가게 되고, ((++a) < 0 ? -(++a) : (++a))으로 바뀌게 된다. ++a 연산이 한번 보다 많이 수행될 수 있다는 문제가 생긴다. ++a가 적절하게 동작하기 위해선 '++a'연산을 ABS(a) 이전에 수행하면 된다. #define SQR x*x 매크로의 경우에는 SQR(a+1)을 하면 a+1*a+1이 되어 의도하지 않은 계산 결과가 나올 수 있다. 이렇게 매크로에서 side effect가 나오게 사용한다면, 의도하지 않은 결과가 나오는 문제가 발생할 수 있어 선택하였다. (L3, P3에 해당하는 위험 정도)

3.2 Visual studio extension에 적용

위에서 조사한 코딩 패턴들이 작성 중인 코드에 존재하는지 찾기 위해, 정규 표현식을 사용하였다. 정규 표현식으로 적절하지 않은 코딩 패턴들, 주의해야 할 코딩 패턴들을 저장하고, 코드를 스캔하여 이 정규 표현식과 매칭되는지 확인하고, 매칭되면 해당 코딩 패턴에 표시하였다. 그리고 표시된 곳에 마우스를 올리면 해당되는 CERT 코딩 패턴 번호와 설명을 보여주도록 하였다.

구현은 microsoft github에 있는 vscode-extension-samples를 기반으로 구현하였다. 이 오픈소스를 기반으로 아래와 같은 정규 표현식을 여러 개 추가하고 vscode api 문서를 참고하여 extension을 수정, 확장 하였다.

if 문의 조건에 = 연산자가 사용된 경우를 찾는 정규 표현식이다. 코드 전체에서 찾아야 하므로 g 옵션을 넣었다. 이러한 정규 표현식을 여러개 작성하여 앞에서 조사한 코딩 패턴들을 찾을 수 있도록 하였다.

```
let regEx = /if\([^=!<>\r\n]*=[^=\r\n]+[^\r\n]*\)/g;
```

그리고 microsoft에서 제공한 extension의 작동 방식인 코드의 변화가 있을 때마다 검사하는 동작 방식

이 코드를 작성할 때 방해가 되는 것 같아, 코드를 작성하고 extension을 실행했을 때 코드를 검사하도록 동작 방식을 바꾸었다.

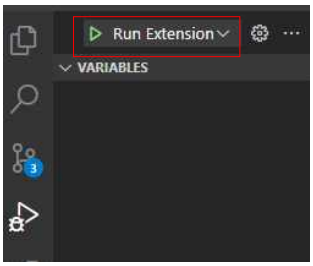
4. 실험결과(선택적)

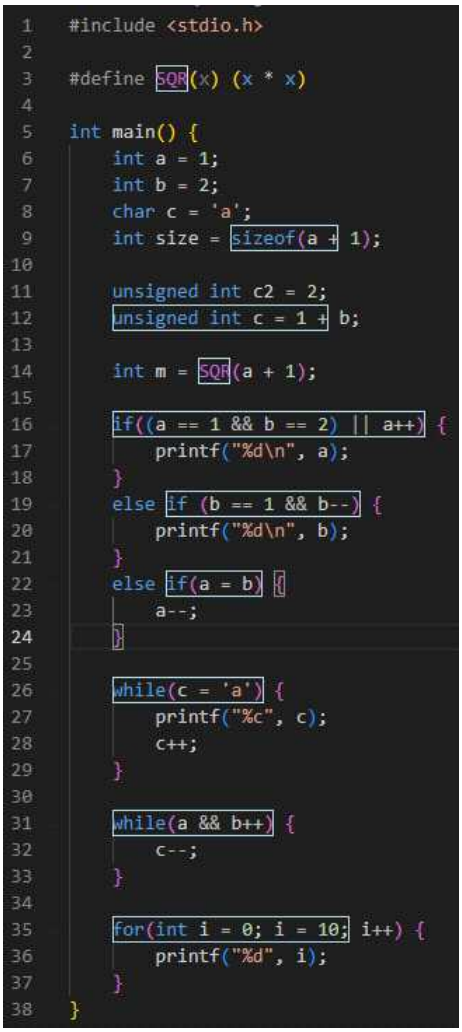
4.1 동작 확인

구현한 extension을 vscode에서 동작해 보았다. 분석 대상 코드는 아래와 같다.

```
#include <stdio.h>
#define SQR(x) (x * x)
int main() {
    int a =1;
    int b =2;
    char c ='a';
    int size =sizeof(a +1);
    unsigned int c2 =2;
    unsigned int c =1 +b;

    int m =SQR(a +1);
    if((a ==1 &&b ==2) ||a++) {
        printf("%d\n", a);
    }
    else if (b ==1 &&b--) {
        printf("%d\n", b);
    }
    else if(a =b) {
        a--;
    }
    while(c ='a') {
        printf("%c", c);
        c++;
    }
    while(a &&b++) {
        c--;
    }
    for(int i =0; i =10; i++) {
        printf("%d", i);
    }
}
```

아래는 만든 extension을 'Run Extension' 해본 결과이다. 'Run Extenson' 키를 눌러서 분석 대상이 되는 코드를 선택하고 'ctrl+shift+p'를 눌러 만든 extension을 클릭하면 결과를 볼 수 있다. 만든 extension의 이름은 'CLint' 이다.





```
#include <stdio.h>

#define SQR(x) (x * x)

int main() {
    int a = 1;
    int b = 2;
    char c = 'a';
    int size = sizeof(a + 1);

    unsigned int c2 = 2;
    unsigned int c = 1 + b;

    int m = SQR(a + 1);

    if((a == 1 && b == 2) || a++) {
        printf("%d\n", a);
    }
    else if (b == 1 && b--) {
        printf("%d\n", b);
    }
    else if(a = b) {
        a--;
    }

    while(c = 'a') {
        printf("%c", c);
        c++;
    }

    while(a && b++) {
        c--;
    }

    for(int i = 0; i = 10; i++) {
        printf("%d", i);
    }
}
```

extension에서 정규 표현식으로 작성한 부분과 일치하는 코드가 네모 박스로 표시되었다. 그리고 해당 코드에 마우스를 올리면 아래와 같이 어떤 코딩 패턴을 위배했는지, 주의해야 할지 정보를 알려준다.

```
(EXP30-C)(shortcut evaluation) If Right-hand side of '||', '&&' has side effect, it
may not be executed
while(a && b++) {
    c--;
}
```

```
(EXP45-C)Recommend to change '=' to '=='
for(int i = 0; i = 10; i++) {
    printf("%d", i);
}
```

```
(INT30-C) Ensure that unsigned integer operations do not wrap. You can check
by using limits.h header and UINT_MAX
unsigned int c = 1 + b;
```

```
(EXP44-C) Do not rely on side effects in operands to sizeof
sizeof(a + 1);
```

```
char c =
int size
unsigned
unsigned
int m = SQR(a + 1);
```

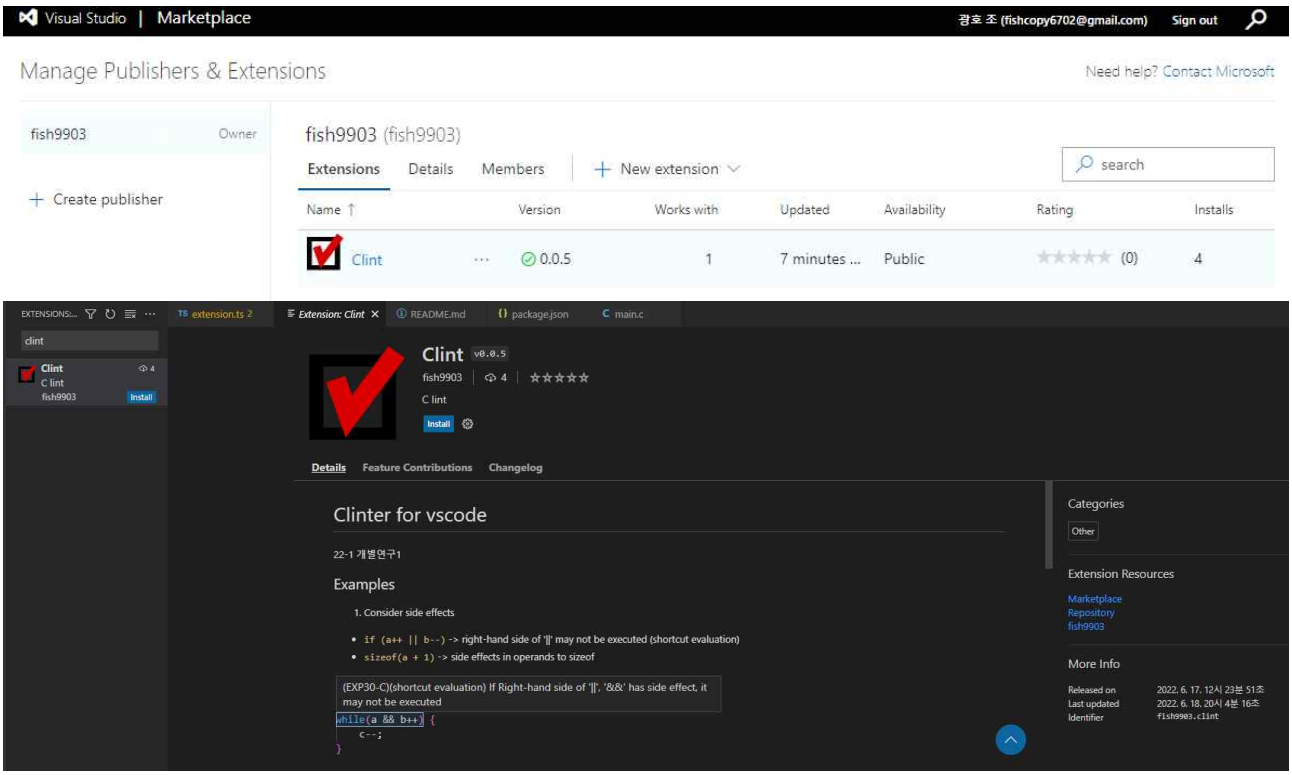
```
#define SQR(x) (x * x)
다음으로 확장:
(a + 1 * a + 1)
(PRI31-C) Avoid side effects in arguments to unsafe macros
```

4.2 extension 배포 및 확인

이렇게 구현한 extension을 샤스트라 웹 IDE에서 실행해보기 위해 visual studio marketplace에 만든 extension을 배포하고 이를 웹 IDE에서 다운받으려 하였다.

Extension 배포는 성공적으로 되었다. 그리고 배포할 때 github 저장소와 연결하고 extension 설명에서 알 수 있도록 하여 이후 extension의 변경을 통해 이번 연구에서 적용한 코딩 패턴 이외의 코딩 패턴을

사용자가 임의로 정규 표현식을 이용해 작성하여 기능을 확장할 수 있도록 하였다.



vscode에서 만든 extension을 검색해 찾은 모습이다. 이것을 설치하여 vscode에서 사용할 수 있다.

하지만 샤스트라 웹 IDE의 vscode에서는 배포한 이 extension을 찾을 수 없어 웹 IDE에서는 사용해보지 못했다.

5. 결론 및 향후 연구

이렇게 특정한 코딩 패턴을 정규 표현식으로 나타내고 이 패턴이 작성 중인 코드에 존재하는지 검사하고 패턴과 일치하면 정보를 알려주는 extension(plug-in)을 만들어 보았다. Extension을 만들면서 코드를 작성할 때 주의해야 할 코딩 패턴들에 대해 공부할 수 있었다.

아쉬웠던 점은 CERT Secure Coding Standard에서는 다양하고 많은 코딩 패턴들을 소개하고 있는데, 이번 extension에서는 많이 반영하지 못했다. 이는 시간상의 문제라기보단 CERT Secure Coding Standard에서 소개하는 코딩 패턴들을 정규 표현식으로 표현할 방법을 찾지 못했기 때문이라고 생각한다. 코딩 패턴들 중 정규 표현식으로 표현가능한 패턴들만 찾고 적용하다 보니 생긴 문제이다. 이렇게 코딩 패턴들을 찾으면서 정규 표현식 만으로는 모든 패턴들을 표현하기 힘들 것 같다는 생각이 들었고, 정규 표현식 말고 다른 방법도 찾아보면 좋았을 것이라는 생각이 들었다.

향후 연구에서는 CERT Secure Coding Standard에서 소개하는 코딩 패턴들을 정규 표현식을 사용하거나 다른 방법을 사용해 표현하고, extension에 적용하는 방향으로 진행하면 좋을 것 같다. extension의 github 저장소를 extension 설명란에 공유하여 코딩 패턴들을 추가하거나 정규 표현식 이외의 방법으로

코딩 패턴을 검사하는 방법이 있으면 기능이 더욱 확장될 것이다. 그리고 사용자가 github에 있는 파일을 보면서 CERT Secure Coding Standard에 없는 코딩 패턴을 정규 표현식으로 나타내어 extension 파일을 수정하면 사용자가 검사하고 싶은 코딩 패턴 또한 검사하도록 extension을 확장할 수 있다.

참고문헌

참고한 오픈소스
microsoft의 vscode extension 예시
https://github.com/microsoft/vscode-extension-samples
https://github.com/microsoft/vscode-extension-samples/tree/main/decorator-sample

참고 문서
vscode extension api 정보, 만드는 법
https://code.visualstudio.com/api/get-started/your-first-extension
https://code.visualstudio.com/api/references/vscode-api

정규 표현식 테스트
https://regexr.com/6l3p1

코딩 규칙 참고 (CERT Secure Coding Standard)
https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards

배보한 extension github
https://github.com/fish9903/Clint