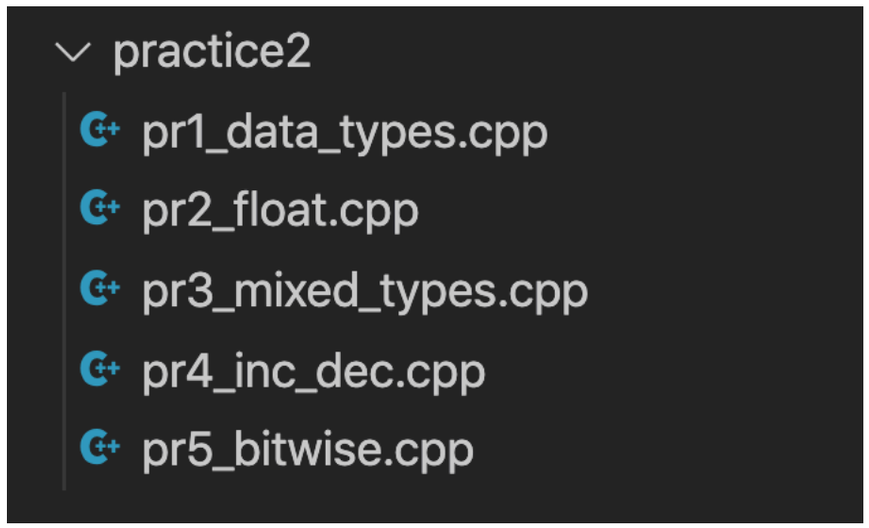
C++ 실습 2

실습 내용

- 이번 주 실습은 이론시간에 배웠던 데이터형과 연산자 사용 예시 등을 직접 코드로 확인해 보는 것입니다.

- 총 5개의 cpp파일로 이루어져 있고, 순차적으로 실행해서 결과를 확인하면 됩니다.



실습 방법

- 실습시에는 다음의 두 가지 방법 중 하나로 진행할 것을 권장합니다.

    1. 실습용으로 제시된 예제 코드를 자신의 파일에 직접 쓰고 컴파일 및 실행.
    2. 실습용 코드를 다운받고 코드를 먼저 읽어 보면서 어떤 실행결과가 나올지 예상해 본 후, 실제 실행결과와 비교.

- 주의) 어떤 방법으로 하든, 반드시 코드만을 보고 실행결과를 예상하는 연습을 해야합니다 - 프로그래밍 연습 및 시험 대비를 위해 매우 중요!

실습파일을 받는 방법

- 실습을 위한 파일들은 강의 GitHub에 공개되어 있습니다.
- 이 공개된 저장소를 그대로 복제해서 가져올 수 있습니다.
- 복제는 Git이라는 소스관리 프로그램의 Clone 명령을 통해서 하게 됩니다.
- 이를 위해서는 먼저 Git이 여러분의 PC에 설치되어 있어야 합니다.

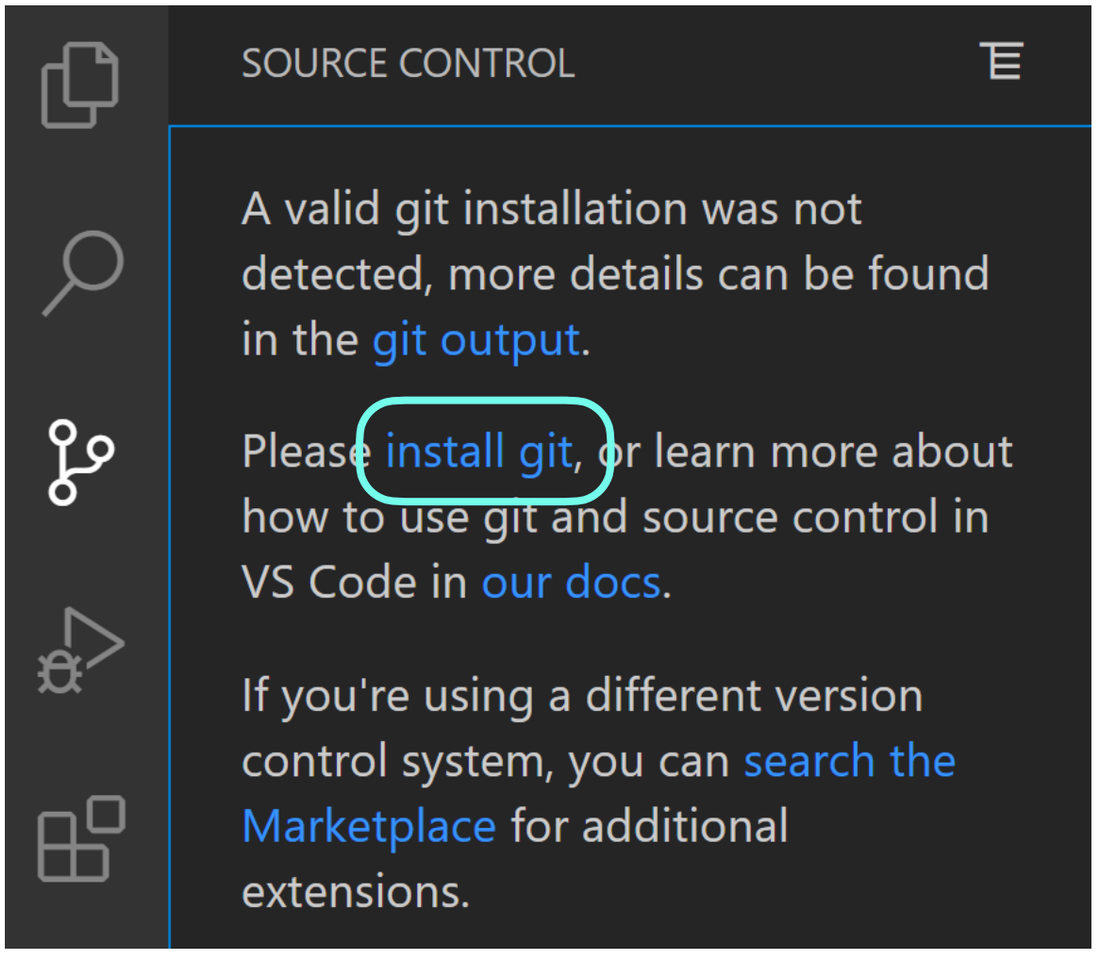
Install Git

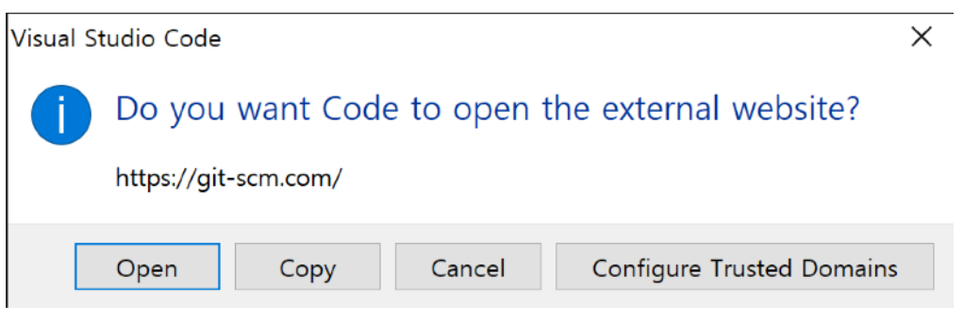
- VSCode의 왼쪽 아이콘 중 세번째를 선택하면, 오른쪽과 같이 설명이 나옵니다.

- 이미 Git이 설치되어 있는 경우, 다른 메시지가 표시되니 Git설치 부분은 건너뛰면 됩니다.

- ‘install git’이라고 표시된 부분을 클릭하면 설치할 수 있는 링크로 연결됩니다.

- 또는 직접 git-scm.com으로 가서 다운받아 설치해도 됩니다.

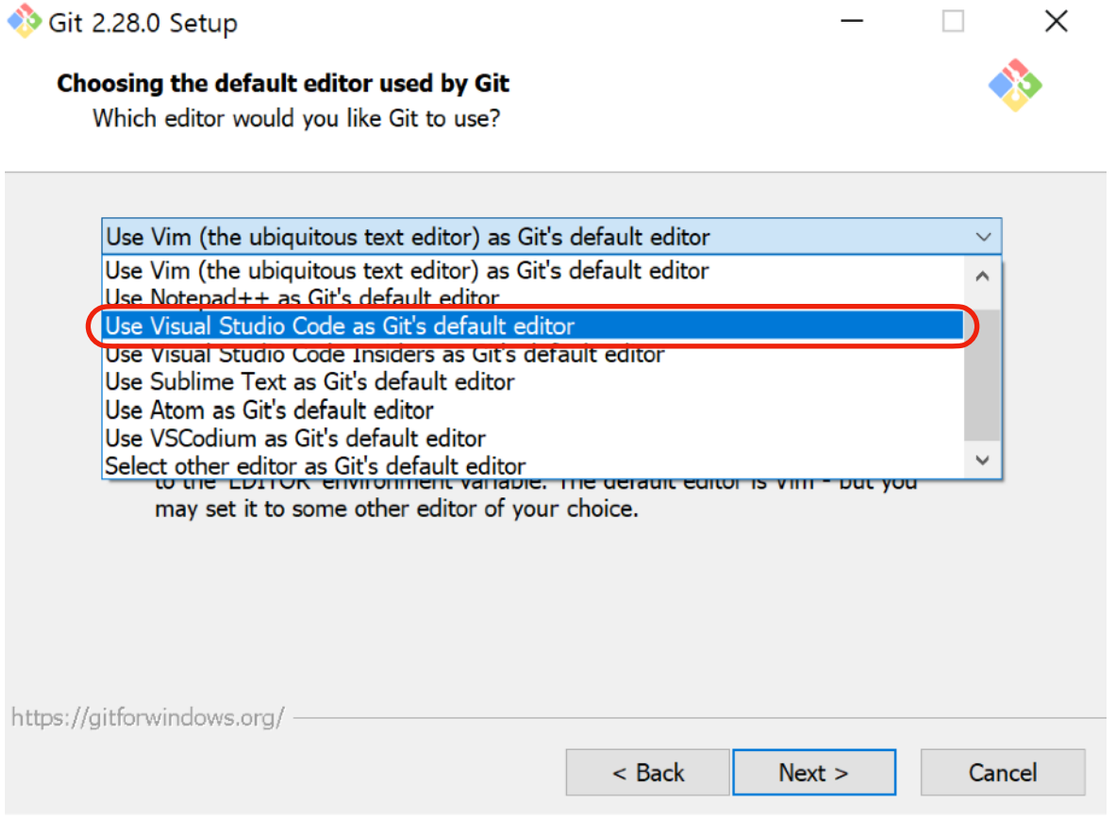


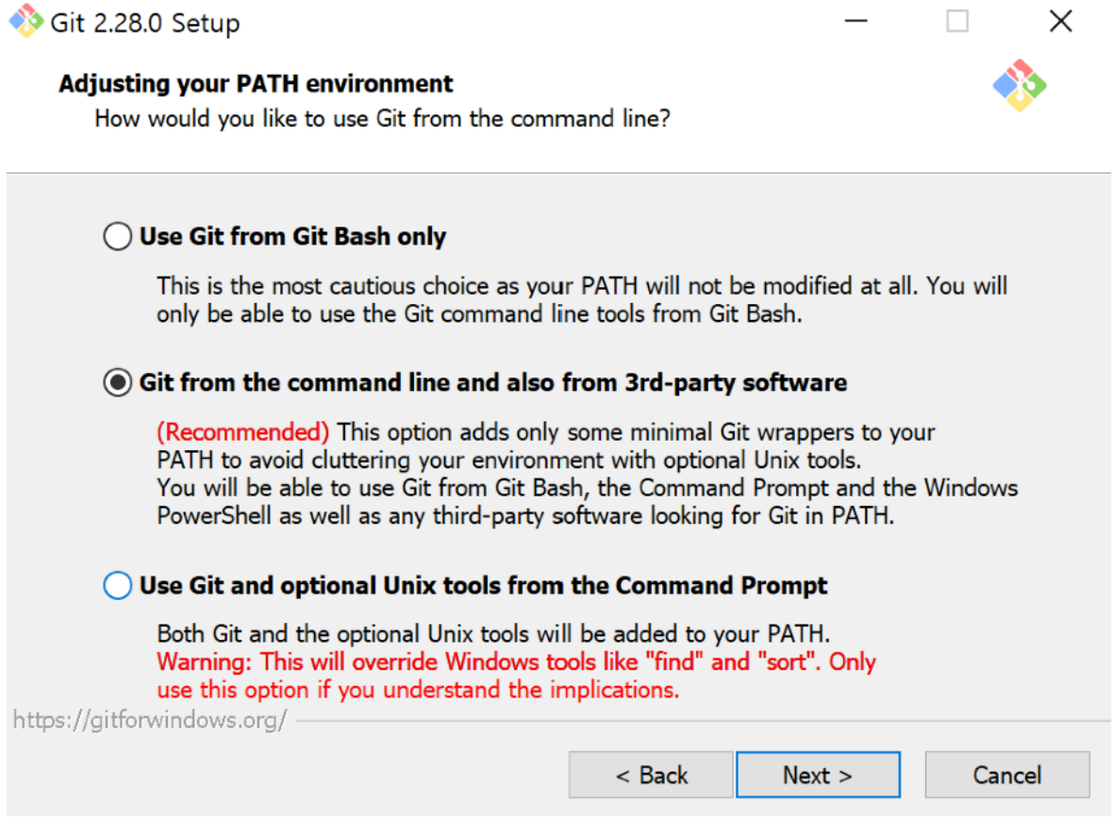


Install Git

- 다양한 선택사항이 나올텐데, 대부분 기본으로 선택되어 있는 것을 고르면 됩니다.

- 아래의 두 가지만 표시된 것을 골랐는지 확인하세요.



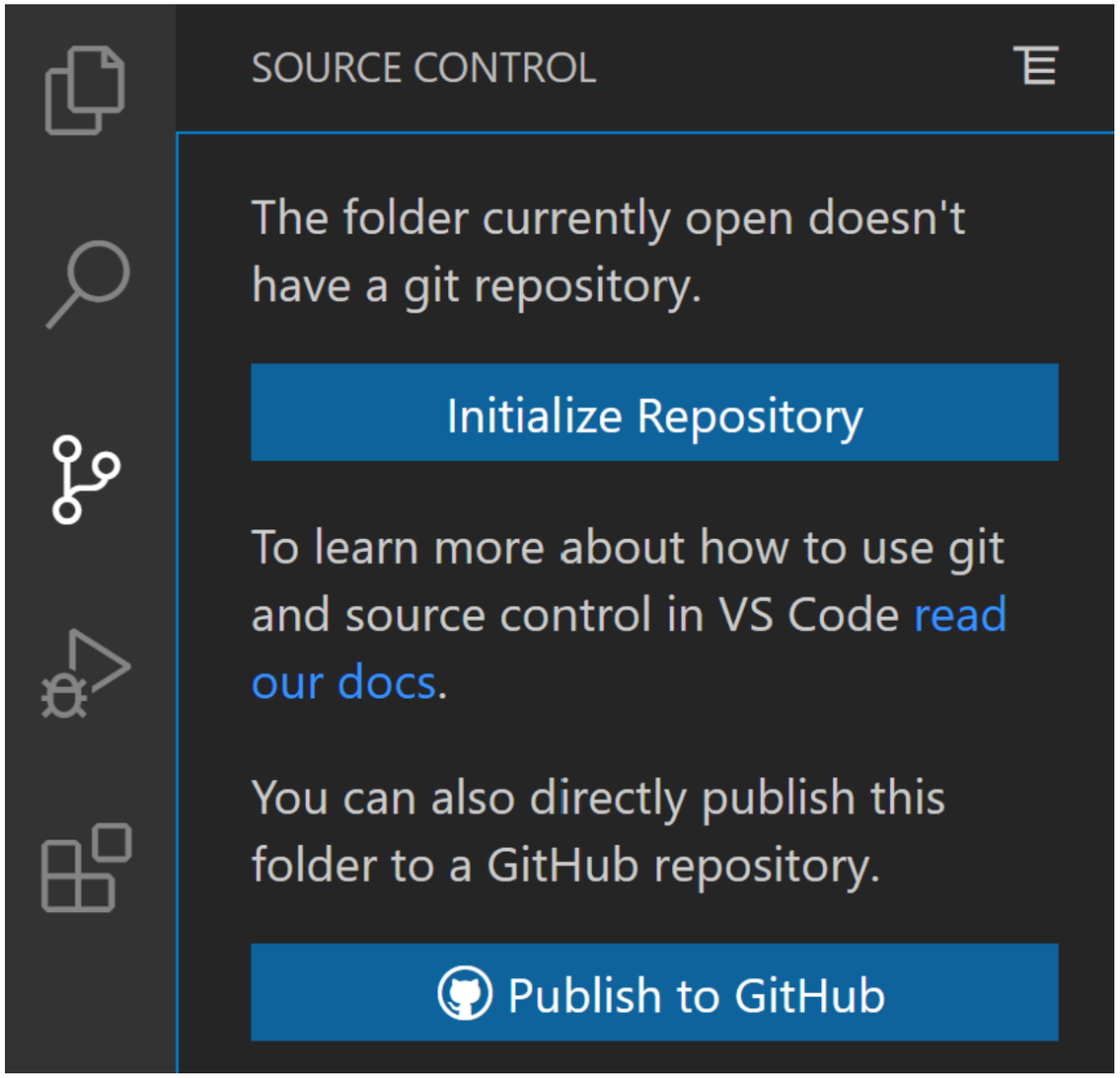


설치 완료 후

- 설치가 잘 완료되었으면, VSCode를 재시작합니다.

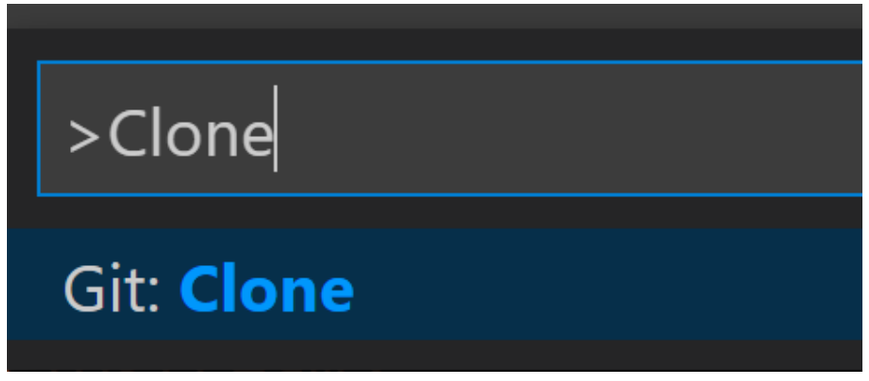
- 그러면 세번째 메뉴 선택시 오른쪽과 같은 화면이 보입니다.

- 설치가 잘 되었고, VSCode가 Git이 설치된 것을 인식한 것이 확인되면 다음 단계로 넘어갑니다.



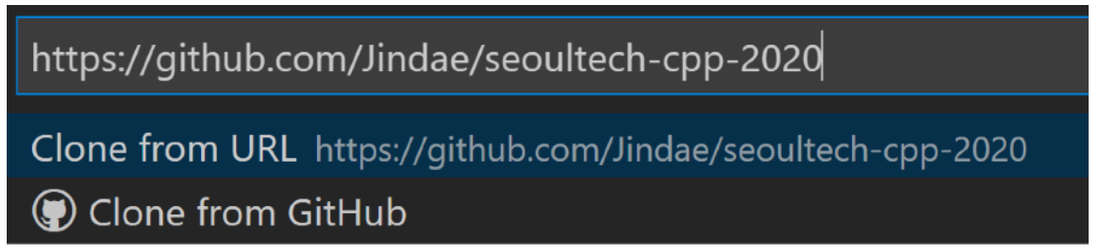
Clone the Repository

- Ctrl (or Cmd)+ Shift + P, 또는 View > Command Palette를 선택하여 오른쪽 첫번째와 같은 입력창을 엽니다.

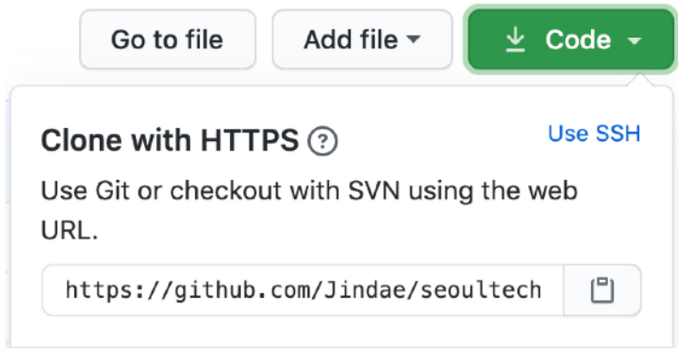


- Clone을 치면 Git: Clone 명령이 검색됩니다.

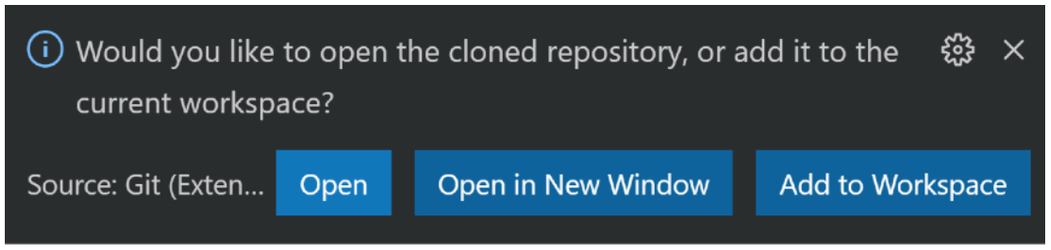
- 선택하면 두번째와 같이 입력창이 나오고, 주소를 입력하면 Clone from URL xxx가 나타납니다.



- 주소는 직접 입력하거나, 강의 GitHub에서 Code버튼을 눌러 복사할 수 있습니다.



Clone from URL



- 주소를 입력하고 명령어를 실행하면, 복제할 저장소(repository)의 위치를 선택하는 화면이 나옵니다.

- 여러번이 선택한 위치에 저장소 이름의 폴더가 생성되고 내용이 복사됩니다.

- 이 과정에서 위와 같은 선택지가 나타나는데, Add to workspace를 선택하여 현재 workspace에 추가하거나, Open in New Window를 선택하여 새 창에서 저장소를 열 수 있습니다.

- 새 창에서 연 경우, 설정을 위해 .vscode 폴더를 복사하는 것을 잊지 마세요.

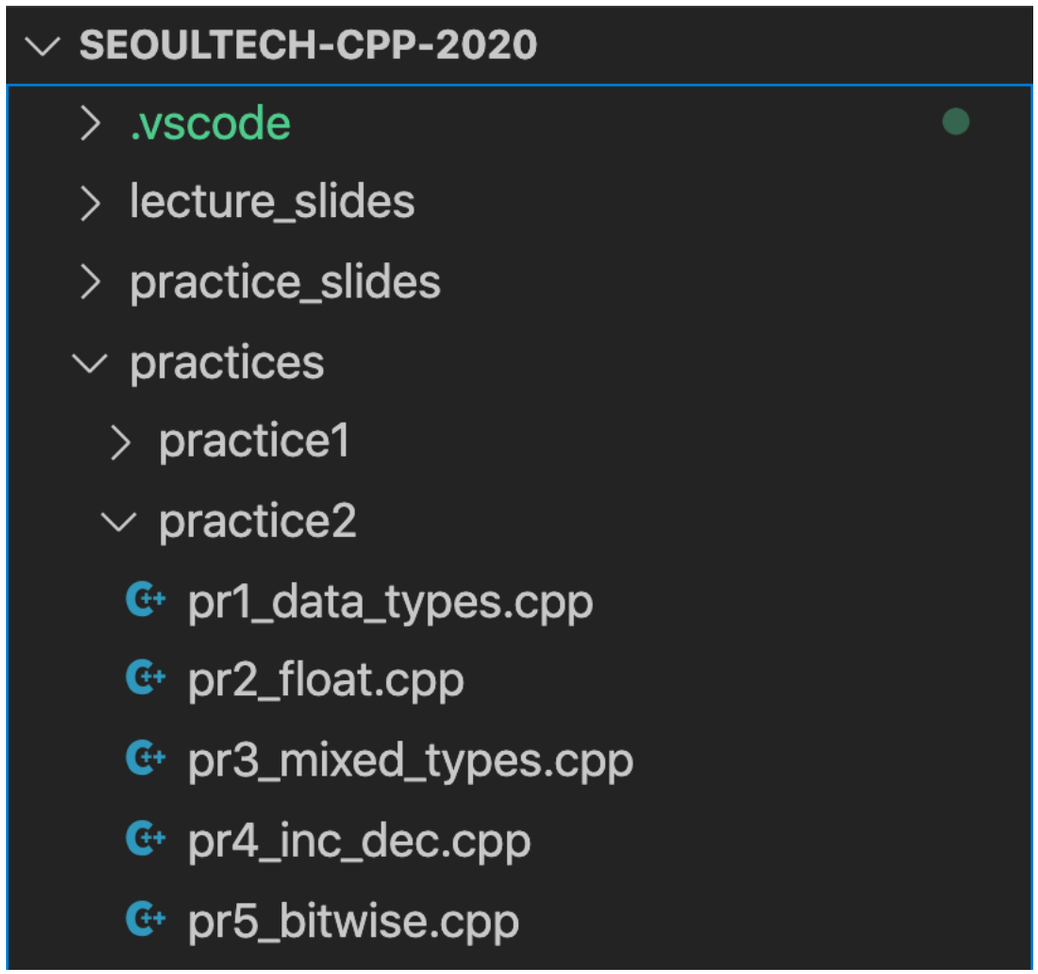
복사 완료 후

- 새 창에서 연 경우의 예시 화면입니다.

- .vscode를 복사하여 설정을 옮겨왔습니다. 초록색으로 표시되는 것은 새로 추가되었다는 걸 인식해서 입니다.

- practices > practice2를 선택하여 펼치면 다음과 같이 5개의 cpp 파일을 볼 수 있습니다.

- 이를 하나씩 실행하여 결과를 확인하면 실습완료입니다.



본격적인 실습에 앞서

- 다시 한 번 강조하지만, 앞으로 소개할 코드 및 설명들을 보고, 실제 파일들을 실행하면 어떤 결과가 나올지 먼저 예측해 보세요.

- 잘 모르겠으면 강의 슬라이드를 다시 참고하세요. 대부분의 예제는강의 슬라이드에서 사용한 것들입니다.

- 이후 파일들을 컴파일하고 실행하여 예측한 결과와 일치하는지 확인하세요.

- 만약 예측이 틀렸다면 무엇을 잘못 생각했는지 다시 한 번 확인하도록 합니다.

- 이 과정은 앞으로 여러분이 실제 프로그래밍을 하고 버그를 고칠 때 무수히 반복하게 될 과정입니다. 빠르게 경험을 쌓기 위해 미리 연습해 두는 것이 좋습니다.

첫번째 실습

- pr1_data_types.cpp 파일을 실행합니다.
- a에는 65가 대입되었고, x, y에는 a가 대입되므로 세 변수는 모두 같은 값을 갖게 됩니다.
- 같은 값의 변수들이 데이터형에 따라 어떻게 다르게 출력되는지 확인해 봅니다.
- std::boolalpha는 bool형 값을 true/false로 출력해주기 위한 것입니다.
- 이 옵션이 없으면 어떻게 출력되는지도 한 번 확인해 보세요.

```cpp
int a = 65;
char x = a;
bool y = a;

cout << a << endl;
cout << x << endl;
cout << std::boolalpha; //boolalpha to print 'true/false'
cout << y << endl;
cout << std::noboolalpha << y << endl; //switch off.

//There are 26 letters + 6 special chracters in the middle.
x = x + 32;
cout << x << endl;
```

첫번째 실습

- 아래는 ASCII코드를 이용해 문자들이 어떻게 표시되는지 보여주는 예제입니다.

- 각각의 숫자가 코드표의 특정 문자에 대입되므로, char형의 변수에 숫자를 더해주는 것으로 다른 문자로 바꿀 수 있습니다.

- ASCII 코드표는 인터넷에서 쉽게 찾을 수 있습니다.

  - https://ko.wikipedia.org/wiki/ASCII

```
int a = 65;
char x = a;
bool y = a;

cout << a << endl;
cout << x << endl;
cout << std::boolalpha; //boolalpha to print 'true/false'
cout << y << endl;
cout << std::noboolalpha << y << endl; //switch off.

//There are 26 letters + 6 special chracters in the middle.
x = x + 32;
cout << x << endl;
```

두번째 실습

- 첫번째 부분은 각 데이터형별로 유효자리수를 표시하는 코드입니다.
- 이론시간에 얘기한 것처럼, 시스템별로 그 숫자가 다를 수 있습니다.
- 두번째 부분은 정밀도에 따라 float가 표현할 수 있는 한계를 보여줍니다.
- c와 d는 모두 a와 b에 1.0을 더한 값입니다.
- 따라서 밑의 계산식은 수학적으로는 모두 결과가 1이 되어야 합니다.
- 실제 어떻게 출력되는지 확인해 보세요.

```
cout << DBL_DIG << endl;
cout << FLT_DIG << endl;
cout << LDBL_DIG << endl;

float a = 1.23E+20f; //1230....0
float b = 1.23E+6f; //1,230,000
float c = a + 1.0f;
float d = b + 1.0f;

cout << "c - a = " << c - a << endl;
cout << "d - b = " << d - b << endl;
```

두번째 실습

- 첫번째 부분은 float형과 double형의 정밀도 차이로 실수가 어떻게 다르게 표시되는지 보여줍니다.

- 실제 pi와 dbl_pi에는 같은 값이 대입되었습니다.

- fixed는 실수를 고정소수점(1.000000와 같은 형식)으로 표시하기 위한 명령입니다.

- setprecision(n)은 소수점이하 n자리까지 표시해주기 위한 명령입니다.

```
float pi = 3.14159265359f;
cout << fixed;
cout << setprecision(6) << pi << endl;
cout << setprecision(15) << pi << endl;

double dbl_pi = 3.14159265359;
cout << setprecision(6) << dbl_pi << endl;
cout << setprecision(15) << dbl_pi << endl;

double x = 0.3;
double y = 0.1 + 0.1 + 0.1;
bool eql = x == y;
cout << boolalpha << "x == y ? " << eql << endl;
cout << setprecision(20);
cout << "x = " << x << endl;
cout << "y = " << y << endl;
```

두번째 실습

- 두 번째 부분은 부동소수점 비교에서 정밀도에 따른 문제점을 보여주기 위한 예시입니다.
- x와 y에 대입되는 수식을 보면, 두 변수는 수학적으로는 같은 값을 가져야 합니다.
- 실제 x == y를 bool형으로 저장하여 결과를 확인해 봅니다.
- 그 아래부분은 소수점 20자리까지 x,y를 표시하여 줍니다.
- 정밀도 범위를 벗어난 부분이 어떻게 달라질 수 있는지 확인해 보세요.

```
float pi = 3.14159265359f;
cout << fixed;
cout << setprecision(6) << pi << endl;
cout << setprecision(15) << pi << endl;

double dbl_pi = 3.14159265359;
cout << setprecision(6) << dbl_pi << endl;
cout << setprecision(15) << dbl_pi << endl;

double x = 0.3;
double y = 0.1 + 0.1 + 0.1;
bool eql = x == y;
cout << boolalpha << "x == y ? " << eql << endl;
cout << setprecision(20);
cout << "x = " << x << endl;
cout << "y = " << y << endl;
```

세번째 실습

- 세번째 실습은 서로 다른 데이터형의 변수로 연산을 실시했을 때의 결과를 확인해보는 것입니다.

- 정수형(int)와 실수형(double)을 섞어 나눗셈을 했을 때, 데이터형에 따라 결과가 어떻게 달라지는지 확인해보세요.

- 결과가 저장되는 변수의 데이터형과, 연산에 사용되는 변수의 데이터형에 주의해서 결과를 확인합니다.

```
int a = 5, b = 2, c = 0;
double d = 0.0;

//direct int division
cout << 5 / 2 << endl;
cout << a / b << endl;

//int division: assign to int
c = a / b;
cout << c << endl;

//int division: assign to double
d = a / b;
cout << d << endl;

//mixed division: int / double
d = a / d;
cout << d << endl;

d = 3 / 2;
cout << d << endl;
```

세번째 실습

- 다음 부분은 각 데이터형별로 표현할 수 있는 숫자의 범위가 다른 것을 확인 해보기 위한 예제입니다.

- 첫번째 부분은 short형이 표현할 수 있는 범위를 넘어선 값이 저장되면 어떻게 되는지 보여줍니다.

- 두번째 부분은 정밀도가 다른 float와 double에 연산 결과를 저장했을 때 얻을 수 있는 결과를 보여줍니다.

- 마지막 부분은 정수형인 y를 형 변환(type cast)하여 실수형인 double로 계산되도록 하는 예제입니다.

```
int y = 40000;
short x = y + y;
cout << x << endl;

double i = 1.0000002;
float j = i * 2.0;
double k = i * 2.0;
cout << fixed << setprecision(7) << j << endl;
cout << k << endl;

i = 5 / double(y);
cout << i << endl;
```

네번째 실습

- 증가/감소 연산자를 사용할 때 전위와 후위표현을 사용하는 것의 차이점을 확인하는 예제입니다.

- 어느 시점에 실제 증가/감소가 일어나는지 출력시점과 맞추어 확인해 보기 바랍니다.

```cpp
int a = 3;
double b = 3.141592;

//increment
cout << "Increment example" << endl;
cout << "a = " << a << " b = " << b << endl;
cout << "a = " << ++a << " b = " << ++b << endl; //Increased before used.
cout << "a = " << a++ << " b = " << b++ << endl; //Not Increased yet.
//Increased at this point.
cout << "a = " << a << " b = " << b << endl;

//decrement
cout << endl << "Decrement example" << endl;
cout << "a = " << a << " b = " << b << endl;
cout << "a = " << --a << " b = " << --b << endl; //Decreased before used.
cout << "a = " << a-- << " b = " << b-- << endl; //Not Decreased yet.
//Decreased at this point.
cout << "a = " << a << " b = " << b << endl;

int c = 1;
int d = --c + c++ + c-- + ++c;
cout << "d = " << d << endl;
```

다섯번째 실습

- 다섯번째 실습은 비트 연산자(bitwise operator)의 사용에 대해 확인해 보는 것입니다.

- std::bitset<8>(a)는 a변수에 저장된 값에서 오른쪽부터(rightmost) 8개의 비트를 출력하는 명령어입니다.

- 각각의 명령어 실행결과와 강의 슬라이드에서 나왔던 예시와 비교해 보세요.

```
int a = 48, b = 37;
int c = 0;
c = a << 1;
cout << "a = " << std::bitset<8>(a) << endl; //bitset<8> to print 8 bits.
cout << "b = " << std::bitset<8>(b) << endl << endl;

//Bitwise NOT
c = ~a;
cout << " a = " << std::bitset<8>(a) << endl;
cout << "~a = " << std::bitset<8>(c) << endl << endl;

//Bitwise AND / OR
c = a & b;
cout << "    a = " << std::bitset<8>(a) << endl;
cout << "    b = " << std::bitset<8>(b) << endl;
cout << "a & b = " << std::bitset<8>(c) << endl << endl;
c = a | b;
cout << "    a = " << std::bitset<8>(a) << endl;
cout << "    b = " << std::bitset<8>(b) << endl;
cout << "a | b = " << std::bitset<8>(c) << endl << endl;
```

```
//Bitwise XOR
c = a ^ b;
cout << "    a = " << std::bitset<8>(a) << endl;
cout << "    b = " << std::bitset<8>(b) << endl;
cout << "a ^ b = " << std::bitset<8>(c) << endl << endl;

//Shift Operators.
cout << "     a = " << std::bitset<8>(a) << endl;
cout << "a << 1 = " << std::bitset<8>(c) << endl;
c = a >> 2;
cout << "     a = " << std::bitset<8>(a) << endl;
cout << "a >> 2 = " << std::bitset<8>(c) << endl << endl;
```

다섯번째 실습

- 후반의 예제 중 첫번째 부분은 2의 보수 표현을 사용하는 signed형과 그렇지 않은 unsigned형의 차이를 보여주는 부분입니다.

- 동일한 비트가 각각 어떤 값으로 나타나는지 확인해 보세요.

```
//signed and unsigned.
c = 1 << 31;
unsigned int d = 1 << 31;
cout << "  signed 1 << 31 = " << c << endl;
cout << "  signed 1 << 31 = " << bitset<32>(c) << endl;
cout << "unsigned 1 << 31 = " << d << endl;
cout << "unsigned 1 << 31 = " << bitset<32>(d) << endl;
```

- 두번째 부분은 일반적인 곱하기/나누기와 bitwise operator로 shift연산을 수행할 때의 성능차이를 보여줍니다.

- clock()은 그 시점의 시각을 저장하고, 둘의 차이를 초단위로 바꾸기 위해 CLOCKS_PER_SEC로 나누어 주었습니다.

```
//Performance comparison
clock_t s_time = clock();
int x = 1;
for(int i=0; i<100000000; i++) {
    x = x << 1; // x * 2
    x = x >> 1; // x / 2
}
clock_t e_time = clock();
cout << fixed << setprecision(6);
cout << "Bitwise Op exec. time = " << (double)(e_time-s_time)/CLOCKS_PER_SEC << "s" << endl;

s_time = clock();
x = 1;
for(int i=0; i<100000000; i++) {
    x *= 2;
    x /= 2;
}
e_time = clock();
cout << "Arithmetic exec. time = " << (double)(e_time-s_time)/CLOCKS_PER_SEC << "s" << endl;
```

실습 제출물

- 출석인정을 위해서, 마지막 다섯번째 실습 파일의 코드를 실행합니다.

- 제일 마지막 부분의 실행시간이 자신의 PC에서 얼마나 걸렸는지 표시해 주는 부분을 캡쳐하여 제출하면 됩니다.

```
  signed 1 << 31 = -2147483648
  signed 1 << 31 = 10000000000000000000000000000000
unsigned 1 << 31 = 2147483648
unsigned 1 << 31 = 10000000000000000000000000000000
Bitwise Op exec. time = 0.249000s
Arithmetic exec. time = 0.282000s
```

실습 정리

- 총 5개 파일에 걸쳐 좀 긴 실습을 진행하였습니다.
- 초반에 몇 번 강조한 것처럼 꼭 코드만을 보고 실행결과를 예측하는 연습을 하세요.
- 나중에 코드만 제시되더라도 코드를 실행하지 않고도 결과를 예측하여 쓸 수 있어야 합니다.