1. 문제 정의

문제는 사칙연산(더하기, 빼기, 곱하기, 나누기)을 각각 수행하는 클래스를 구현하고, 사용자가 입력한 연산자에 따라 해당 클래스를 이용하여 연산을 수행하는 프로그램을 작성하는 것이다. 사용자는 두 개의 정수와 연산자를 입력하고, 입력한 연산자에 따라 더하기, 빼기, 곱하기, 나누기 중 하나의 연산을 수행하는 프로그램을 요구한다. 나눗셈의 경우 0으로 나누는 예외 상황도 처리해야 한다.

2. 문제 해결 방법

문제를 해결하기 위해 주어진 요구 사항을 명확히 분석하고, 객체지향 프로그래밍의 원칙을 적용하여 구조적으로 문제를 풀어나가는 방법을 선택했다. 이 문제는 입력된 두 수와 연산자에 따라 사칙연산을 각각 수행해야 하므로, 각 연산을 독립된 클래스로 설계하여 문제를 해결하는 것이 효과적이다.

1. 클래스 설계

연산을 담당하는 각 기능을 분리하기 위한 Add, Sub, Mul, Div라는 네 개의 클래스는 각기 다른 연산을 수행하며, 공통적인 구조를 갖는다.

- 각 클래스는 두 개의 피연산자(a와 b)를 저장하는 멤버 변수를 가지고 있다.
- 두 수를 입력받아 멤버 변수에 저장하는 setValue() 메서드를 제공한다.
- 연산을 수행하고 그 결과를 반환하는 calculate() 메서드를 제공한다.

2. 클래스 구조

이 프로그램에서 사용되는 모든 연산은 두 개의 정수를 입력받아 처리되므로, 각 클래스는 공통적으로 다음의 기능을 갖도록 한다.

피연산자 저장 - 각 클래스는 정수형 멤버 변수 a, b를 가진다. 이를 통해 사용자로부터 입력받은 두 수를 저장한다.

setValue() 메소드 - 두 수를 입력받아 a와 b 멤버 변수에 저장한다. 이 함수는 모든 클래스에서 동일한 방식으로 작동하므로, 사용자가 입력한 값을 각 클래스에 동일한 방식으로 전달할 수 있다.

calculate() 메소드 - 각 클래스는 해당 연산을 수행하고 결과를 반환하는 calculate() 메서드를 가진다. 이 메서드는 클래스마다 서로 다른 방식으로 연산을 수행한다.
- Add: a + b를 계산한다.
- Sub: a - b를 계산한다.

- Mul: a * b를 계산한다.
- Div: a / b를 계산하며, b가 0일 경우 예외 처리를 한다.

3. 메인 프로그램 설계

메인 프로그램에서는 사용자가 입력한 연산자에 따라 적절한 클래스를 선택하고, 해당 클래스의 setValue()와 calculate() 메서드를 호출하여 연산을 수행한다.

입력 처리 - 프로그램은 무한 루프를 사용하여 두 개의 정수와 연산자를 입력받는다. 입력된 연산자에 따라 덧셈, 뺄셈, 곱셈, 나눗셈 중 하나의 연산을 수행하게 된다.

조건에 따른 클래스 선택 - 입력된 연산자에 따라 Add, Sub, Mul, Div 클래스를 선택한다.

if 조건문을 사용하여 입력된 연산자가 어떤 연산을 나타내는지 확인하고, 해당하는 연산 클래스를 선택한다. 예를 들어, 사용자가 +를 입력하면 Add 클래스의 객체를 생성하고, 두 수를 설정한 뒤 덧셈을 수행한다.

4. setValue()와 calculate() 메서드 호출

각 클래스의 객체를 생성한 후, setValue() 메서드를 호출하여 두 개의 정수를 클래스 내부의 a, b에 저장한다. 그런 다음, 해당 객체의 calculate() 메서드를 호출하여 연산을 수행하고 결과를 반환받는다.

setValue() 호출 - setValue() 메서드는 두 개의 입력된 값을 객체의 멤버 변수에 복사하는 역할을 한다.
calculate() 호출 - calculate() 메서드는 해당 클래스가 처리하는 연산을 수행하고, 그 결과를 반환한다.

5. 나눗셈 예외 처리

나눗셈(Div)의 경우, 0으로 나누는 경우에 대한 예외 처리가 필요하다. Div 클래스의 calculate() 메서드에서 b가 0일 경우 경고 메시지를 출력하고, 결과를 0으로 반환하는 방식으로 예외 처리를 한다.

```
if (b == 0) {
    cout << "0으로 나눌 수 없습니다." << endl;
    return 0;
}
```

이 예외 처리를 통해 사용자가 잘못된 입력을 했을 때도 프로그램이 비정상적으로 종료되지

않고, 사용자에게 메시지를 제공하면서 프로그램이 계속 실행될 수 있도록 했다.

6. 무한 루프 및 프로그램 종료

프로그램은 무한 루프를 통해 계속해서 사용자로부터 입력을 받는다. 사용자가 연산을 마친 후에도 프로그램은 종료되지 않고 다음 연산을 입력받는다. 특정 조건에서 프로그램을 종료하려면 추가적인 종료 조건을 설정할 수 있지만, 이 문제의 요구사항에서는 종료 조건 없이 무한히 동작하는 구조로 설계되어 있다.

```
while (true) {
    cout << "두 정수와 연산자를 입력하세요 : ";
    int x, y;
    char op;
    cin >> x >> y >> op;
```

이 while문을 사용한 코드를 통해 프로그램은 사용자가 연산을 계속할 수 있고, 매번 입력을 받고 결과를 출력한 후에도 종료되지 않고 다시 입력을 받을 수 있다.

7. 사칙연산 연산자 외 연산자 처리

사용자가 +, −, *, / 이외의 연산자를 입력했을 때, 프로그램은 "잘못된 연산자입니다."라는 메시지를 출력하고 다시 입력을 요청한다. 이를 통해 사용자의 잘못된 입력에 대한 오류 처리를 간단히 수행했다.

3. 아이디어 평가

1. 클래스 설계

각 연산을 독립적인 클래스로 분리한 객체지향 설계는 코드의 재사용성과 유지보수성을 크게 향상시켰다. 각 클래스가 독립적으로 기능을 수행하므로, 연산 방식에 대한 수정이 필요할 때 해당 클래스만 수정하면 된다.

결과 : 유지보수가 쉬워졌으며, 새로운 연산 기능을 추가할 때 다른 클래스와 충돌 없이 독립적으로 추가할 수 있는 구조가 완성되었다. 이는 프로그램의 확장성을 높이는 데 중요한 역할을 한다.

2. 클래스 구조

각 클래스가 명확한 역할을 하고 있어 코드의 가독성이 좋아졌다. Add, Sub, Mul, Div 클래스를 통해 각 연산이 구분되었으며, 코드의 흐름을 쉽게 이해할 수 있다. 또한, 클래스 간의 메서드 사용 방식이 일관되어 있어 코드 전체의 일관성도 확보되었다.

결과 : 가독성이 향상되었으며, 이후 코드를 분석하거나 다른 개발자가 유지보수를 진행할 때도 코드를 쉽게 이해할 수 있게 되었다.

3. 메인 프로그램 설계

설계의 확장성을 평가해 보면, 새로운 연산을 추가할 때도 기존의 구조를 그대로 활용할 수 있다. 제곱 연산이나 나머지 연산 등을 추가할 때는 클래스를 추가로 정의하고, calculate() 메서드에 필요한 연산을 구현하면 된다. 기존의 클래스나 로직을 수정하지 않고도 기능을 확장할 수 있다.

결과 : 새로운 기능을 추가하는 과정에서 코드 수정이 거의 필요 없으며, 추가적인 연산 클래스만 만들면 된다. 이 구조는 향후 프로그램의 확장성에 긍정적인 영향을 미쳤다.

4. setValue()와 calculate() 메서드 호출

setValue()와 calculate() 메서드를 사용하는 구조는 코드의 재사용성, 유지보수성, 명확성을 크게 향상시켰다. setValue()는 입력 값을 멤버 변수에 저장하고, calculate()는 연산을 수행해 결과를 반환하는 방식으로, 입력과 연산이 명확히 분리되었다.

결과 : 입력 처리와 연산 로직이 독립적으로 유지되어 가독성과 오류 추적이 용이해졌다. 또한, 동일한 인터페이스를 사용해 각기 다른 연산을 처리할 수 있어 확장성이 뛰어나고, 새로운 연산을 쉽게 추가할 수 있다. 예외 처리도 명확히 구분되었으며, 사용자에게 직관적인 사용법을 제공하여 사용자 편의성을 높였다.

5. 나눗셈 예외 처리

나눗셈에서 0으로 나누는 경우를 처리한 예외 처리 로직은 프로그램의 안정성을 높였다. 사용자가 잘못된 입력을 했을 때, 프로그램이 비정상적으로 종료되지 않고 메시지를 출력하며 계속 실행될 수 있게 하는 방식은 안정성 확보에 도움을 주었다.

결과 : 예외 처리가 없었더라면 0으로 나누기 시 프로그램이 비정상 종료되었을 가능성이 있지만, 오류가 발생하더라도 프로그램이 안전하게 계속 실행되어 프로그램의 안정성 확보에 중요한 역할을 했다.

6. 무한 루프 및 프로그램 종료

프로그램은 사용자가 연산을 수행할 때마다 계속해서 새로운 입력을 받을 수 있도록 무한 루프를 적용했다. 이를 통해 사용자는 여러 번의 연산을 입력할 수 있으며, 잘못된 입력이 발생하더라도 프로그램이 종료되지 않고 계속 동작할 수 있다.

결과 : 연산자가 잘못 입력되거나 예외 상황이 발생하더라도 프로그램이 종료되지 않고 다시 입력을 받도록 설계되어 있어, 사용자 편의성과 프로그램의 안정성이 증가하였다.

7. 사칙연산 연산자 외 연산자 처리

잘못된 연산자가 입력될 경우 "잘못된 연산자입니다."라는 메시지를 출력하도록 하여 사용자에게 명확한 피드백을 제공한다. 사용자가 올바른 연산자를 입력할 때까지 프로그램이 다시 입력을 요청하기 때문에, 오류에 대한 대응이 가능하다.

결과: 입력 값이 잘못되었을 때 사용자가 무엇을 잘못했는지 명확하게 인지할 수 있었으며, 이를 통해 잘못된 입력을 다시 수정하고 프로그램을 계속 사용할 수 있었다.

4. 문제를 해결한 키 아이디어 또는 알고리즘 설명

문제를 해결하기 위한 핵심 아이디어는 객체지향 프로그래밍을 이용하여 연산 기능을 독립된 클래스로 분리하는 방식이다. 각각의 연산을 수행하는 클래스를 설계함으로써, 프로그램의 유지보수성, 재사용성, 확장성을 극대화하고, 연산 처리 흐름을 단순화할 수 있었다.

1. 클래스별 연산 분리 및 독립성

각 연산은 Add, Sub, Mul, Div라는 독립적인 클래스로 구현되었으며, 모든 클래스는 공통된 구조를 가진다. 이로 인해 각 연산 클래스는 다른 클래스에 영향을 주지 않고 독립적으로 작동할 수 있다. 각 클래스는 피연산자를 저장하는 setValue() 메서드와, 연산을 처리하는 calculate() 메서드를 포함하고 있어 연산 과정이 캡슐화되었다. 이 구조는 연산을 처리하는 로직을 명확히 구분하며, 특정 연산에 문제가 생겨도 다른 연산에는 영향을 미치지 않는다.

2. 공통 인터페이스 적용

모든 연산 클래스가 동일한 형태의 메서드를 제공하므로, 공통 인터페이스를 통해 다양한 연산을 처리할 수 있다. 이 아이디어는 메인 함수에서 입력된 연산자에 따라 적절한 연산 클래스를 선택하고, 동일한 방식으로 setValue()와 calculate() 메서드를 호출함으로써 코드의 일관성을 유지한다. 사용자가 어떤 연산을 입력하더라도 처리 방식이 일관되며, 입력 처리와 연산 수행이 구체적으로 분리되었다.

3. 입력과 연산의 분리

각 클래스는 입력된 두 숫자를 setValue() 메서드를 통해 저장하고, 이 값을 기반으로 calculate() 메서드에서 연산을 수행한다. 이렇게 입력과 연산을 명확히 분리한 것은 코드의

구조적 명확성을 높이고, 이후 유지보수 및 오류 추적을 용이하게 만든다. 특히, 입력 로직이 연산 로직과 독립적으로 작동하므로, 새로운 연산을 추가하더라도 기존 코드의 수정이 최소화된다.

4. 예외 처리 및 안정성

Div 클래스에서는 0으로 나누기에 대한 예외 처리를 구현하여, 프로그램의 안정성을 보장하였다. 사용자가 잘못된 입력을 했을 경우 프로그램이 비정상적으로 종료되지 않고, 메시지를 출력한 후에도 계속 실행될 수 있도록 설계되었다. 이 예외 처리 방식은 프로그램의 안정성을 향상시키는 중요한 요소였다.

5. 확장성 및 유연성

이 설계 방식은 프로그램의 확장성을 향상시킨다. 새로운 연산이 추가될 때 기존 코드를 수정할 필요 없이, 새로운 연산 클래스를 추가하고 calculate() 메서드를 구현하기만 하면 된다. 이러한 구조 덕분에 프로그램의 유연성이 높아졌으며, 다양한 연산 요구 사항에 손쉽게 대응할 수 있게 되었다. 이는 향후 프로그램을 확장하거나 기능을 추가할 때 매우 중요한 이점이다.

6. 무한 루프를 통한 연속 입력 처리

main() 함수에서 무한 루프를 사용해 사용자가 계속해서 연산을 입력할 수 있도록 설계했다. 이를 통해 프로그램은 매번 종료되지 않고 새로운 입력을 받아 연속적인 연산을 수행할 수 있다. 이 방식은 사용자 편의성을 증대시키며, 잘못된 입력이 발생해도 프로그램이 종료되지 않고 다시 입력을 요구한다.

코드 실행 흐름

이 문제의 코드는 객체지향 프로그래밍을 활용하여 덧셈, 뺄셈, 곱셈, 나눗셈을 각각 독립된 클래스로 구현한 후, 사용자의 입력에 따라 해당 연산을 수행하는 프로그램이다.

1. main() 함수 실행

프로그램이 시작되면 `main()` 함수가 실행된다. `main()` 함수에서는 사용자로부터 연산을 위한 입력을 받는 무한 루프를 실행한다.

```
int main() {
    while (true) {
        int x, y;
        char o;
        cout << "두 정수와 연산자를 입력하세요 (+, -, *, /): ";
```

```
cin >> x >> y >> o;
```

- 무한 루프 : `while (true)`는 사용자가 프로그램을 종료하지 않는 한 계속해서 입력을 받도록 한다. 이 루프는 프로그램이 종료되지 않고 연속적으로 여러 번의 연산을 처리할 수 있게 한다.
- 입력받기: `cin >> x >> y >> op;`를 통해 사용자는 두 개의 정수 `x`와 `y`, 그리고 연산자 `op`를 입력한다. 사용자가 "3 4 +"를 입력하면 두 정수는 각각 `x = 3`, `y = 4`, 연산자는 `op = +`로 저장된다.

2. 입력된 연산자에 따라 클래스 선택

입력된 연산자(`op`)에 따라 적절한 연산 클래스를 선택한다. 각각의 클래스는 해당 연산을 수행하는 기능을 가진다. 조건문을 사용하여 어떤 연산을 할지 결정하고, 그에 맞는 클래스를 호출한다.

```
if (o == '+') {
    Add a;
    a.setValue(x, y);
    cout << "결과: " << a.calculate() << endl;
}
else if (o == '-') {
    Sub s;
    s.setValue(x, y);
    cout << "결과: " << s.calculate() << endl;
}
else if (o == '*') {
    Mul m;
    m.setValue(x, y);
    cout << "결과: " << m.calculate() << endl;
}
else if (p == '/') {
    Div d;
    d.setValue(x, y);
    cout << "결과: " << d.calculate() << endl;
}
else {
    cout << "잘못된 연산자입니다." << endl;
}
```

1. 연산자 조건 검사 : `if-else` 문을 사용하여 입력된 연산자가 각각 `+`, `-`, `*`, `/`인지 확인한다.
2. 클래스 객체 생성 : 연산자에 맞는 클래스의 객체를 생성한다.

- `Add add;`: 덧셈을 처리할 `Add` 클래스의 객체를 생성한다.
- `Sub sub;`: 뺄셈을 처리할 `Sub` 클래스의 객체를 생성한다.
- `Mul mul;`: 곱셈을 처리할 `Mul` 클래스의 객체를 생성한다.
- `Div div;`: 나눗셈을 처리할 `Div` 클래스의 객체를 생성한다.

3. setValue() 호출 : 각 연산 클래스의 `setValue()` 메서드를 호출하여 입력된 두 정수를 클래스 멤버 변수에 저장한다. 예를 들어, `add.setValue(x, y);`는 `x`와 `y`를 `Add` 클래스의 멤버 변수 `a`와 `b`에 저장한다.
4. calculate() 호출 : `calculate()` 메서드를 호출하여 연산을 수행하고 결과를 반환한다. 예를 들어, `add.calculate()`는 `Add` 클래스에서 `a + b` 연산을 수행하여 결과를 반환한다.
5. 결과 출력 : `cout`을 통해 연산 결과를 화면에 출력한다.

예외 처리

- 만약 연산자가 `+`, `-`, `*`, `/`가 아닌 경우, `else` 구문에서 "잘못된 연산자입니다."라는 메시지를 출력하고 루프를 다시 시작하여 새로운 입력을 받는다.

3. 각 클래스의 동작

각 연산 클래스(`Add`, `Sub`, `Mul`, `Div`)는 모두 동일한 구조로 동작하지만, 연산 로직만 다르게 구현되어 있다.

Add 클래스 (덧셈)

```
class Add {
    int a, b;
public:
    void setValue(int x, int y);
    int calculate();
};

void Add::setValue(int x, int y) {
    a = x;
    b = y;
}

int Add::calculate() {
    return a + b;
}
```

- calculate() : 멤버 변수 `a`와 `b`의 합을 계산하고 그 값을 반환한다.

Sub 클래스 (뺄셈)

```
class Sub {
    int a, b;
public:
    void setValue(int x, int y);
    int calculate();
};

void Sub::setValue(int x, int y) {
    a = x;
    b = y;
}

int Sub::calculate() {
    return a - b;
}
```

- calculate() : `a - b` 연산을 수행한다.

Mul 클래스 (곱셈)

```
class Mul {
    int a, b;
public:
    void setValue(int x, int y);
    int calculate();
};

void Mul::setValue(int x, int y) {
    a = x;
    b = y;
}

int Mul::calculate() {
    return a * b;
}
```

- calculate() : `a * b` 연산을 수행한다.

Div 클래스 (나눗셈 및 예외 처리)

```
class Div {
    int a, b;
public:
    void setValue(int x, int y);
    int calculate();
};

void Div::setValue(int x, int y) {
    a = x;
    b = y;
}

int Div::calculate() {
    if (b == 0) {
        cout << "0으로 나눌 수 없습니다." << endl;
        return 0;
    }
    return a / b;
}
```

- calculate() : `a / b` 연산을 수행하지만, `b`가 0일 경우 예외 처리를 통해 "0으로 나눌 수 없습니다." 메시지를 출력하고 결과를 0으로 반환한다.

4. 연속적인 연산 처리

위의 과정을 통해 연산이 완료되면 다시 무한 루프로 돌아가 사용자로부터 새로운 연산을 입력받는다. 프로그램이 종료되지 않고 계속해서 반복적으로 두 정수와 연산자를 입력받고, 연산 결과를 출력하는 방식으로 동작한다.

1. 프로그램 시작 → 무한 루프를 통해 사용자 입력 대기.
2. 두 정수와 연산자 입력.
3. 입력된 연산자에 따라 해당 연산 클래스를 선택.
4. `setValue()`를 통해 두 정수 설정.
5. `calculate()`를 호출하여 연산 수행 후 결과 출력.
6. 다시 루프를 통해 다음 입력 대기.

이 과정을 반복하며 사용자가 원할 때까지 연산을 계속 수행할 수 있다.