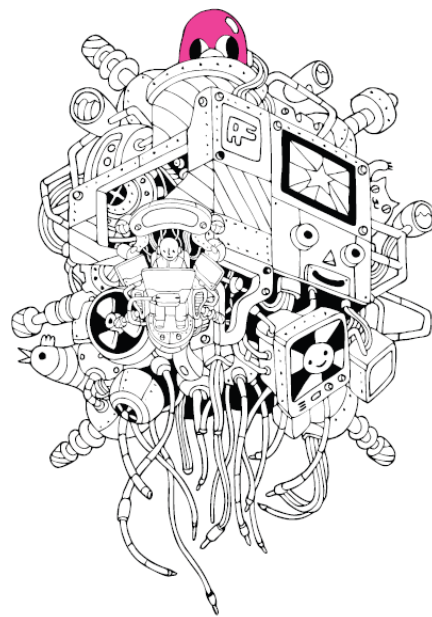


윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 26. 매크로와 선행처리기

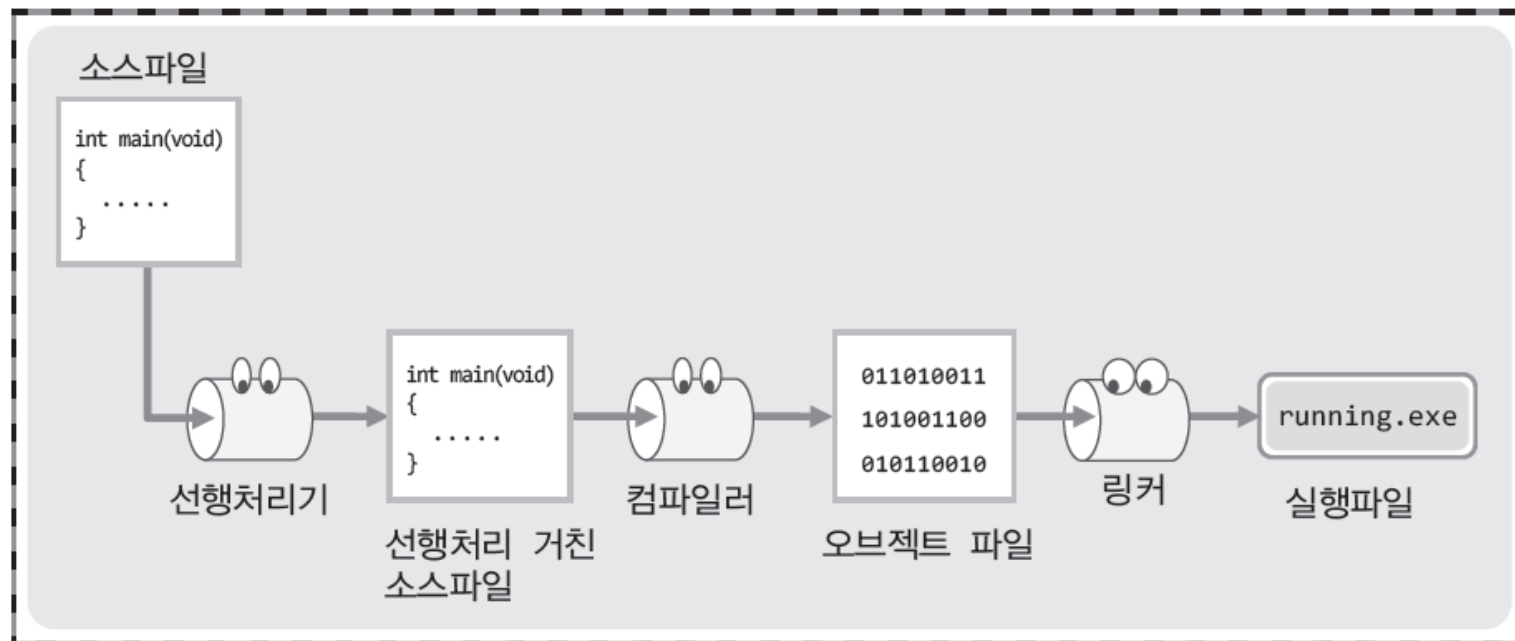
윤성우의 열혈 C 프로그래밍



Chapter 26-1. 선행처리기와 매크로

윤성우 저 열혈강의 C 프로그래밍 개정판

선행처리는 컴파일 이전의 처리를 의미합니다.



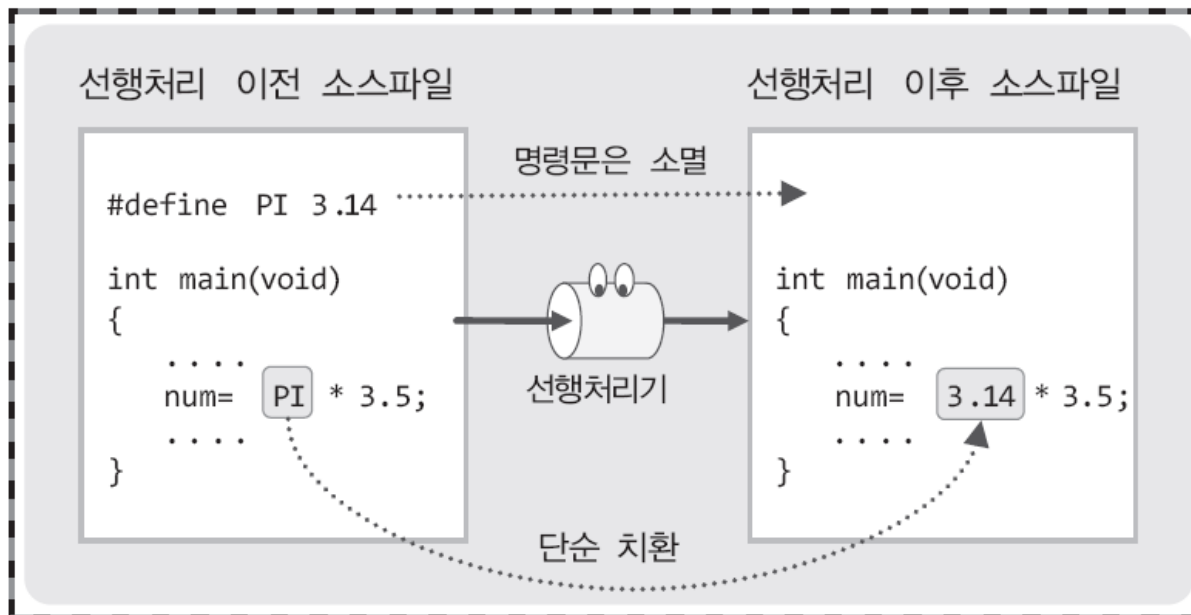
일반적으로 컴파일의 과정에 포함이 되어 이야기하지만, 선행처리의 과정과 컴파일의 과정은 구분이 된다.



선행처리기의 일 간단히 맛보기

#define PI 3.14 가 의미하는 바는 다음과 같다.

“PI를 3.14로 치환하라!” .



컴파일러에 비해서 선행처리기의 역할은 매우 간단하다. 쉽게 말해서 ‘단순한 치환’의 작업을 거친다. 그리고 선행처리기에게 무엇인가를 명령하는 문장은 #으로 시작한다.

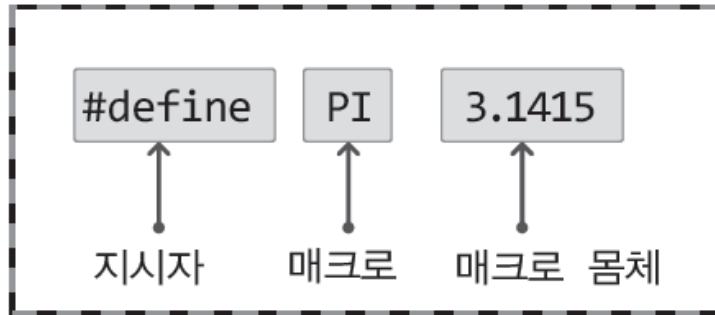
윤성우의 열혈 C 프로그래밍



Chapter 26-2. 대표적인 선행처리 명령문

윤성우 저 열혈강의 C 프로그래밍 개정판

#define: Object-like macro



PI를 3.1415로 무조건 치환하라!

```

#define NAME      "홍길동"
#define AGE       24
#define PRINT_ADDR puts("주소: 경기도 용인시\n");
int main(void)
{
    printf("이름: %s \n", NAME);
    printf("나이: %d \n", AGE);
    PRINT_ADDR;
    return 0;
}
    
```

```

printf("이름: %s \n", "홍길동");
printf("나이: %d \n", 24);
puts("주소: 경기도 용인시 \n");
    
```

치환된 결과물!

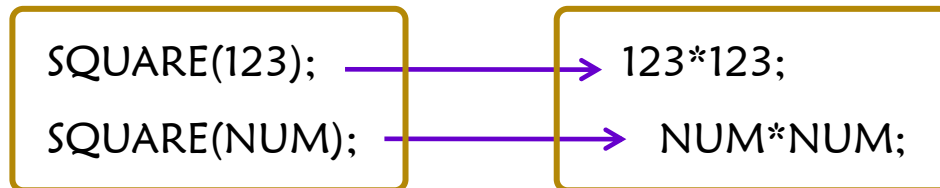
실제 컴파일 되는 문장들!

실행결과

```

이름: 홍길동
나이: 24
주소: 경기도 용인시
    
```

#define: Function-like macro



이러한 변환의 과정을 가리켜
'매크로 확장(macro expansion)'이라 한다.

SQUARE(X) 패턴에 부합!

따라서 치환!

매크로 확장의 예

```
#define SQUARE(X) X*X
int main(void)
{
    int num=20;

    /* 정상적 결과 출력 */
    printf("Square of num: %d \n", SQUARE(num));
    printf("Square of -5: %d \n", SQUARE(-5));
    printf("Square of 2.5: %g \n", SQUARE(2.5));

    /* 비정상적 결과 출력 */
    printf("Square of 3+2: %d \n", SQUARE(3+2));
    return 0;
}
```

```
Square of num: 400
Square of -5: 25
Square of 2.5: 6.25
Square of 3+2: 11
```

실행결과

3과 2의 합인 5가 SQUARE 함수의 인자가 되어, 25가 출력되는 것이 상식적이다! 그러나 출력결과는 다르다!

잘못된 매크로의 정의와 소괄호의 해결책

```
#define SQUARE(X)  X*X
```

SQUARE(3+2)	➔	3+2*3+2	➔	11	오류의 원인
SQUARE((3+2))	➔	(3+2)*(3+2)	➔	25	사용하기 불편

```
#define SQUARE(X)  (X)*(X)
```

SQUARE(3+2)	➔	(3+2)*(3+2)	➔	25	이 경우는 해결
num=120/SQUARE(2)	➔	120 / (2)*(2)			여전히 문제!

```
#define SQUARE(X)  ((X)*(X))
```

num=120/SQUARE(2)	➔	120 / ((2)*(2))		최상의 해결책!
-------------------	---	-----------------	--	----------

매크로를 두 줄에 걸쳐서 정의하는 방법

```
#define SQUARE(X)  
((X)*(X))
```

이렇듯 두 줄에 걸쳐서 매크로를 정의하면 에러가 발생한다!

```
#define SQUARE(X) \  
((X)*(X))
```

첫 번째 줄의 끝에 `\`을 삽입하면 에러가 발생하지 않는다.
이는 `\`이 매크로의 정의가 이어짐을 뜻하기 때문이다.



먼저 정의된 매크로의 사용

```
#define PI 3.14
#define PROUDCT(X, Y) ((X)*(Y))
#define CIRCLE_AREA(R) (PROUDCT((R), (R))*PI)

int main(void)
{
    double rad=2.1;
    printf("반지름 %g인 원의 넓이: %g \n", rad, CIRCLE_AREA(rad));
    return 0;
}
```

반지름 2.1인 원의 넓이: 13.8474

실행결과

위 예제에서 보이듯이, 앞 줄에서 먼저 정의된 매크로는 새로운 매크로를 정의하는데 있어서 사용될 수 있다.

일반함수와 비교한 매크로 함수의 장점

장점 1.

매크로 함수는 일반 함수에 비해 실행속도가 빠르다.

함수의 호출을 완성하기 위해서는 별도의 메모리 공간이 필요하고 호출된 함수로의 이동 및 반환의 과정을 거쳐야 한다. 반면 매크로 함수는 정의된 몸체로 치환이 이뤄지니 이러한 일이 불필요하며, 때문에 실행속도가 빨라질 수 밖에 없다.

장점 2.

자료형에 따라서 별도로 함수를 정의하지 않아도 된다.

전달되는 인자의 자료형에 구분을 받지 않으므로 자료형에 의존적이지 않다.



매크로 함수의 단점

단점

1. 정의하기가 정말로 까다롭다.
2. 디버깅하기가 쉽지 않다. *선행처리 후 컴파일러에 의해서 예러가 감지되므로*

매크로 함수로 정의하면?

```
int DiffABS(int a, int b)
{
    if(a>b)
        return a-b;
    else
        return b-a;
}
```

잘못 정의된 매크로

```
#define DIFF_ABS(X, Y)  ( (x)>(y) ? (x)-(y) : (y)-(x) )
int main(void)
{
    printf("두 값의 차: %d \n", DIFF_ABS(5, 7));
    printf("두 값의 차: %g \n", DIFF_ABS(1.8, -1.4));
    return 0;
}
```

그러나 컴파일러는 컴파일 된 이후의 내용을 기준으로 오류를 이야기한다.

함수를 매크로로 정의하기 위한 조건

조건 1.

작은 크기의 함수

한 두줄 정도 크기의 작은 함수가 아니면 매크로로 정의하는 것이 쉽지 않아서 오류발생 확률이 높아진다. 그리고 if~else, for와 같이 실행의 흐름을 컨트롤 하는 문장도 매크로로 정의하기 쉽지 않다.

조건 2.

호출의 빈도수가 높은 함수

함수를 매크로로 정의하는 데에는 성능적 측면이 고려된다. 그런데 호출의 빈도수가 높지 않으면 애써 매크로로 함수를 정의하는 수고를 할 이유가 줄어든다.



윤성우의 열혈 C 프로그래밍



Chapter 26-3. 조건부 컴파일을 위한 매크로

윤성우 저 열혈강의 C 프로그래밍 개정판

#if . . . #endif : 참이라면

```
#define ADD 1
#define MIN 0

int main(void)
{
    int num1, num2;
    printf("두 개의 정수 입력: ");
    scanf("%d %d", &num1, &num2);
```

```
#if ADD    // ADD가 '참'이라면
    printf("%d + %d = %d \n", num1, num2, num1+num2);
#endif
```

```
#if MIN    // MIN이 '참'이라면
    printf("%d - %d = %d \n", num1, num2, num1-num2);
#endif
```

```
    return 0;
}
```

실행처리 과정에서 *ADD*가 '참' 이
면 ~*endif* 까지 컴파일 대상에 포함

실행처리 과정에서 *MIN*이 '참' 이
면 ~*endif* 까지 컴파일 대상에 포함

실행결과

```
두 개의 정수 입력: 5 4
5 + 4 = 9
```


#ifdef ... #endif : 정의되었다면

```
// #define ADD 1
#define MIN 0
int main(void)
{
    int num1, num2;
    printf("두 개의 정수 입력: ");
    scanf("%d %d", &num1, &num2);
```

ADD와 MIN의 정의 여부만 중요한 상황이라면 각각의 매크로에 값을 지정할 필요가 없다. 즉, 다음과 같이 정의해도 된다.

#define ADD

#define MIN

```
#ifdef ADD // 매크로 ADD가 정의되었다면
    printf("%d + %d = %d \n", num1, num2, num1+num2);
#endif
```

ADD가 정의되었다면 ~endif 까지 컴파일 대상에 포함

```
#ifdef MIN // 매크로 MIN이 정의되었다면
    printf("%d - %d = %d \n", num1, num2, num1-num2);
#endif
```

MIN이 정의되었다면 ~endif 까지 컴파일 대상에 포함

```
    return 0;
}
```

실행결과

```
두 개의 정수 입력: 7 2
7 - 2 = 5
```

#ifndef . . . #endif : 정의되지 않았다면

```
#ifndef ADD      ADD가 정의되지 않았다면 ~endif 까지 컴파일 대상에 포함
    printf("%d + %d = %d \n", num1, num2, num1+num2);
#endif
```

```
#ifndef MIN      MIN이 정의되지 않았다면 ~endif 까지 컴파일 대상에 포함
    printf("%d - %d = %d \n", num1, num2, num1-num2);
#endif
```



#else의 삽입: #if, #ifdef, #ifndef에 해당

```
#define HIT_NUM 5

int main(void)
{
    #if HIT_NUM==5
        puts("매크로 상수 HIT_NUM은 현재 5입니다.");
    #else
        puts("매크로 상수 HIT_NUM은 현재 5가 아닙니다.");
    #endif
    return 0;
}
```

포함 조건

매크로 `HIT_NUM`이 5가 아닐 경우 포함되는 문장

실행결과 매크로 상수 `HIT_NUM`은 현재 5입니다.

`#else`를 추가해서

조건이 ‘참’ 이 아닌 경우에 컴파일의 대상에 포함시킬 문장들을 구성할 수 있다.

#elif의 삽입: #if에만 해당

```
#define HIT_NUM 7

int main(void)
{
    #if HIT_NUM==5
        puts("매크로 상수 HIT_NUM은 현재 5입니다.");
    #elif HIT_NUM==6
        puts("매크로 상수 HIT_NUM은 현재 6입니다.");
    #elif HIT_NUM==7
        puts("매크로 상수 HIT_NUM은 현재 7입니다.");
    #else
        puts("매크로 상수 HIT_NUM은 5, 6, 7은 확실히 아닙니다.");
    #endif

    return 0;
}
```

조건에 따라서 이중에서 하나의 문장만 컴파일의 대상으로 포함
이 된다.

매크로 상수 HIT_NUM은 현재 7입니다.

실행결과

#elif와 #else를 추가해서 if ~ else if ~ else 구문과 동일한 형태를 구성할 수 있다.

윤성우의 열혈 C 프로그래밍



Chapter 26-4. 매개변수의 결합과
문자열화

윤성우 저 열혈강의 C 프로그래밍 개정판

문자열 내에서는 매크로의 매개변수 치환 불가

[문자열 치환을 목적으로 정의된 매크로]

```
#define STRING_JOB(A, B)    "A의 직업은 B입니다."
```

치환의 대상이 되는 *A*와 *B*가 문자열 안에 존재함에 주목하자!



[매크로의 확장 결과]

STRING_JOB(이동춘, 나무꾼) ➡ "A의 직업은 B입니다."

STRING_JOB(한상순, 사냥꾼) ➡ "A의 직업은 B입니다."

이동춘과 나무꾼, 그리고 한상순과 사냥꾼이 기대대로 치환되지 않았다. 문자열 안에서는 치환이 일어나지 않기 때문이다. 이럴 때 고려해 볼 수 있는 연산자가 # 연산자이다!

문자열 내에서 매크로 매개변수 치환: # 연산자

[# 연산자를 이용해서 정의된 매크로]

```
#define STR(ABC) #ABC
```

매개변수 *ABC*에 전달되는 인자를 문자열 "*ABC*"로 치환해라!



[# 연산자 기반의 매크로 확장 결과]

STR(123)



"123"

STR(12, 23, 34)



"12, 23, 34"

연산자를 이용한 문제의 해결!

다음 문장 선언은

```
char * str="ABC" "DEF";
```

다음의 문장 선언과 같음과

```
char * str="ABCDEF"
```

연산자를 근거로 하여 앞서 제시한 문제를 해결한 결과

둘 이상의 문자열 선언을 나란히 하면, 이는 하나의 문자열 선언으로 인식이 된다.

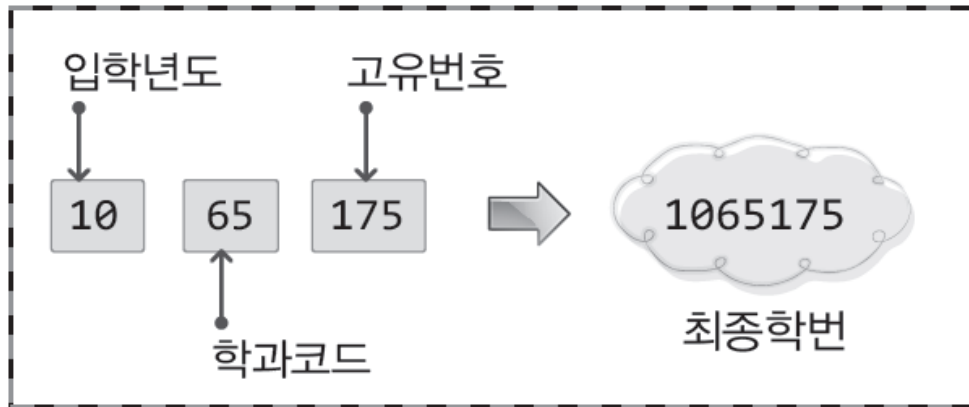


```
#define STRING_JOB(A, B)  #A "의 직업은 " #B "입니다."
int main(void)
{
    printf("%s \n", STRING_JOB(이동춘, 나무꾼));
    printf("%s \n", STRING_JOB(한상순, 사냥꾼));
    return 0;
}
```

실행결과

이동춘의 직업은 나무꾼입니다.
한상순의 직업은 사냥꾼입니다.

매크로 연산자 없이 단순 연결은 불가능



세 개의 숫자를 단순 연결하여 학번을 구성해야 한다고 가정해 보자. 그리고 이를 위한 매크로를 정의해 보자.

```
#define STNUM(Y, S, P)
```

YSP

치환 대상 YSP를 연결해 놓으면 이는 그냥 YSP로 인식이 된다.

```
#define STNUM(Y, S, P)
```

Y S P

치환은 제대로 이뤄지나 Y와 S와 P가 연결되어 있지 않아서 10 65 175 와 같이 치환이 이뤄진다.

```
#define STNUM(Y, S, P)
```

((Y)*100000+(S)*1000+(P))

연산자를 모르는 상태에서의 최선의 해결책!

단순 연결 관련 예제 기반 문제점 확인

```
// #define STNUM(Y, S, P)  YSP
// #define STNUM(Y, S, P)  Y S P
#define STNUM(Y, S, P)  ((Y)*100000+(S)*1000+(P))

int main(void)
{
    printf("학번: %d \n", STNUM(10, 65, 175));
    printf("학번: %d \n", STNUM(10, 65, 075));
    return 0;
}
```

1065175

1065061

실행결과

실행결과를 보면 1065075로 치환이 이뤄지지 않았음을 알 수 있다. 이는 075가 8진수로 해석된 결과이다. 따라서 이 경우에는 다음과 같이 문장을 구성해야 한다.

`printf("학번: %d \n", STNUM(10, 65, 75));`

위 예제에서 보이는 매크로도 좋은 해결책이 되지 못한다. 필요한 것은 '단순 연결'인데, 8진수 인식의 문제로 인해서 생각할 요소가 발생하기 때문이다! 이는 매크로 STNUM의 사용에 있어서 주의를 요하는 요소가 되므로 좋은 매크로 정의라 할 수 없다!

필요한 형태로 단순 결합: ## 연산자

[##연산자를 이용해서 정의된 매크로]

```
#define CON(UPP, LOW) UPP ## 00 ## LOW
```

매개변수 *UPP*와 *LOW*에 달되는 인자를 *UPP00LOW*로 단순히 연결해서 치환해라!



[## 연산자 기반의 매크로 확장 결과]

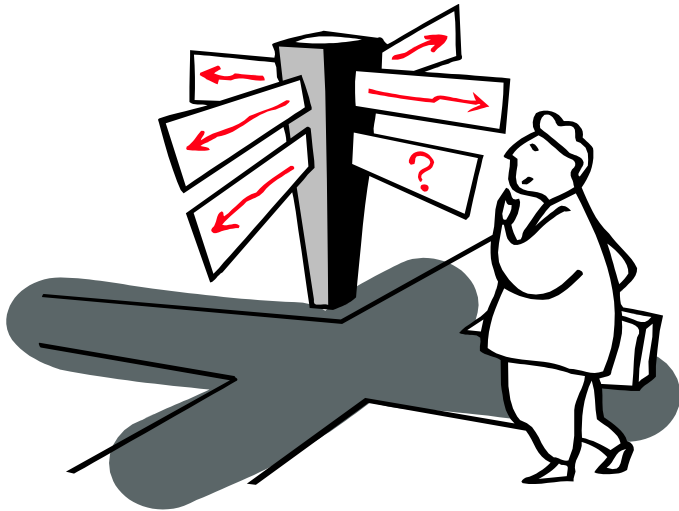
```
int num = CON(22, 77);
```



```
220077
```

이렇듯 ## 연산자를 이용해서 다음과 같이 매크로를 정의해야 학번을 단순치환 할 수 있다.

```
#define STNUM(Y, S, P) Y ## S ## P
```



Chapter 26이 끝났습니다. 질문 있으신지요?