# 객체지향 프로그래밍 노트
# 디파인 매크로(define macro)

강대기*

2007년 10월 10일

**요 약**

본 고에서는 #define 지시어(directive)가 매크로로 사용될 때 유의할 점을 중점적으로 설명하고, #include 지시어(directive)에 대해서 간단히 알아보고, 조건부 컴파일 방법에 대해 서술하겠다.

## 제 1 절 #define 지시어(directive)에 의한 매크로

원래 매크로는 일련의 명령들을 자주 사용해야 할 때, 하나의 키 입력으로 그 일들이 자동으로 수행되게 만들어 사용하기 위한 키 입력을 말한다.

C/C++에선 define에 의해 특정 문자열들이 다른 긴 문자열들로 정의되면, 전처리기가 자동으로 그 특정 문자열을 자동으로 치환할 때, 이를 매크로라고 부른다. 예를 들어 다음과 같이 define으로 선언이 되면, C/C++ 컴파일러가 컴파일하기 전에 전처리기(preprocessor)가 매크로로 PI를 3.14159로 치환 처리한다.

```
#define PI 3.14159
```

초기에 C에서 매크로는 다음의 두가지 주된 이유로 도입되었다.

하나는 일괄적인 관리이다. 예를 들어 프로그램에서 자주 사용되는 테이블의 레코드의 개수가 25000이라고 하자. 전체 레코드를 읽거나 검색할 필요가 있을 때마다 루프를 사용해야 하고 25000이라는 값이 필요하게 된다. 그런데, 나중에 그 값이 바뀌게 되면 프로그램을 찾아다니면서 바꿔줘야 하는 문제가 생긴다. 대신 define으로 맨 위나 특정 헤더 파일에서 다음과 같이 지정해 주면 나중에 편하게 처음 지정한 곳으로 바꾸면 된다.

```
#define TABLE_SIZE 25000
```

C++에서는 이보다 개선된 const가 도입되었다.

두 번째는 실행 속도의 문제로 함수가 들어가야 할 부분에서 함수 대신 실행 코드 부분 자체를 그대로 집어넣기 위해 사용된다. 실제로 C의 getchar나 putchar는 함수가 아니라 매크로이다. 예를 들어 각도를 입력받아 라디안(radian; 호도)으로 바꾸는 매크로는 다음과 같이 정의된다.

```
#define DEGTORAD(x) ((x) / 57.29578)
```

C++에서는 이보다 개선된 inline 함수가 도입되었다.

## 제 2 절 #define 매크로의 문제점

위의 두번째 이유로 매크로를 사용하면서 가장 큰 문제점 중 하나는 프로그래머들이 매크로를 함수로 오인하고 주의없이 사용한다는 것이다. [1]

위의 각도(degree)를 라디안(radian)으로 바꾸는 매크로를 고려해보자. 각도는 우리가 흔히 생활에서 쓰는 90도, 180도 등의 각도이다. 라디안, 즉 호도은 반지름이 1인 단위 원(unit circle)의 중심에서 특정 각으로 만들어지는 호의 길이를 통해 그 각을 나타내는 방법이다. 즉, 360도는 원둘레가 되서 $2\pi$가 되고, 180도는 원둘레의 반이 되서 $\pi$가 된다. 따라서 특정 각도에 대한 호도의 값은 $\pi : 180 = rad : deg$ 라는 공식에 따라 임의의 각도를 받아서 라디안으로 바꾸는 매크로는 다음과 같이 설계할 수 있다.

```
#define DEGTORAD(x) x / 57.29578
```

이는 프로그램에서 예를 들면 다음과 같이 사용될 수 있다.

```
#include <iostream>
using namespace std;
#define DEGTORAD(x) x / 57.29578
int main()
{
  double deg;
  cin>> deg;
  cout << DEGTORAD(deg) << endl; // 1번 예
  cout << DEGTORAD(180) << endl; // 2번 예
  cout << DEGTORAD(179+1) << endl; // 3번 예
  cout << DEGTORAD(180)/DEGTORAD(179+1) << endl; // 4번 예
  return 0;
}
```

위의 1번 예에서 `DEGTORAD(deg)`는 `deg / 57.29578` 로 바뀌게 된다. 예를 들어 deg에 대해 180을 입력하면, $\pi$의 근사치인 3.14159 가 나온다.

그러나 문제는 많은 다양한 표현식이 들어갈 경우, 원했던 결과가 나오지 않을 때가 많다는 것이다. 예를 들어 2번 예의 `DEGTORAD(180)`의 경우에는 `180 / 57.29578`로 바뀌어 $\pi$의 근사치인 3.14159 가 나오는 반면, 3번 예의 경우 `DEGTORAD(179+1)`은 `179 + 1 / 57.29578`로 바뀌어 179.017이 나온다.

이것은 물론 우리가 이 매크로를 만들었을 때 원했던 결과가 아니다. 따라서, 이 문제를 해결하고자 앞의 프로그램에서 다음과 같이 매크로를 바꿀 수 있다.

```
#define DEGTORAD(x) (x) / 57.29578
```

[1] 이 절에서 다루는 주제는 교재 426 427 쪽에서도 소개되어 있다.

이제 문제는 해결되었을까?

다시 실행해보면 3번 예는 이제 `(179 + 1) / 57.29578`이 되어 제대로 3.14159를 출력한다. 그런데, 4번 예의 실행 결과는 1이 나와야 함에도 `0.000304617` 가 나옴을 알 수 있다. 그 이유는 4번 예의 `DEGTORAD(180)/DEGTORAD(179+1)` 가 `(180) / 57.29578 / (179 + 1) / 57.29578` 로 치환되었기 때문이다. 그러나 우리가 원하는 바는 `((180) / 57.29578) / ((179 + 1) / 57.29578)` 이다. 따라서 이를 위해서는 다시 매크로를 다음과 같이 바꾸어야 한다.

```
#define DEGTORAD(x) ((x) / 57.29578)
```

이제 무조건 괄호를 넣는 방법으로 웬만한 문제는 해결될 거처럼 보인다.

그렇다면 두 개의 수를 비교하여 작은 것을 반환하는 다음의 매크로를 보도록 하자.

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

일단 다음과 같이 바꾸어야 할 것이다.

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

그럼에도 불구하고 `MIN(x++,--y)` 의 경우에 대해서는 부수 효과 (side effect) 가 발생하게 된다. 왜냐하면 `MIN(x++,--y)` 가 `((x++)>(--y)?(--y):(x++))` 로 치환되기 때문이다. 이렇게 되면 x와 y의 값이 의도했던 것과는 달리 두 번 증가하게 된다. 여기서 주목할 것은 이러한 부수 효과가 매크로 내에서 변수가 치환될 수 있는 인자가 두 번 나오는 게 문제가 된다는 점이다.

GNU C 와 같은 일부 컴파일러에서는 이러한 문제에 대한 해결책이 존재하긴 하지만, ANSI C의 표준적인 해결책이 아니다. C++에서는 이러한 문제를 근본적으로 인라인(inline) 함수를 통해 해결할 수 있다.

앞의 매크로들에 대한 C++의 인라인 함수들은 다음과 같다.

```
inline double degToGrad(double x) { return x/57.29578; }
inline int min(int a, int b) { return a>b ? b : a; }
```

## 제 3 절 #include 지시어

include 지시어는 뒤에 나오는 헤더 파일의 내용을 전처리기로 하여금, 소스 파일에서 지시어가 있는 위치에 그대로 집어넣게 한다.

예를 들어 다음과 같은 include 지시어는 stdio.h라는 헤더 파일을 집어넣는다.

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

이렇게 헤더 파일을 포함시키는 경우, 헤더 파일 이름이 괄호로 묶여 있으면, 전처리기는 해당 컴파일러의 표준 헤더 파일이 들어있는 디렉토리들에서 그 헤더 파일을 찾는다.

그런데, 괄호가 아니라 다음과 같이 큰 따옴표로 묶으면, 전처리기는 현재의 작업 디렉토리나, 그 include 지시어를 쓰는 소스 파일이 있는 디렉토리에서 그 헤더 파일을 찾는다.

```
#include "bool.h"
```

만일 특정 디렉토리에서 헤더 파일을 찾게 하고 싶다면 다음과 같이 하면 된다.

```
#include "d:/include/myHeader.h"
```

## 제 4 절 조건부 컴파일

다른 전처리기 지시어에 대해 간단히 알아보자.

매크로를 정의하기 위한 지시어로 #define이 있다. 다음 지시어는 LIMIT를 400으로 지정한다.

```
#define LIMIT 400
```

이렇게 정의된 매크로 상수는 #undef 지시어로 정의를 해제할 수 있다.

```
#define LIMIT 400

// LIMIT를 400 으로 쓸 수 있다.

#undef LIMIT

// 이제 LIMIT를 400 으로 쓸 수 없다.
```

그 외에도 #ifdef, #else, #endif 등의 지시어가 있다.

```
#ifdef DONGSEO
#include "dsu.h"
#else
#include "knu.h"
#endif
```

위의 예를 보면, DONGSEO라는 매크로가 정의되어 있으면 “dsu.h” 헤더 파일을 삽입하고, 그렇지 않으면 “knu.h” 라는 헤더 파일을 삽입한다.

또한, #ifdef 와 #endif 는 프로그램에서 다음과 같이 디버깅 출력을 위해 사용될 수도 있다.

```
#define DEBUG
#include <iostream>
using namespace std;
int main (void)
{
  int result = 0;
  for (int i=1;i<=100;i++)
  {
    // 뭔가 중요한 계산
```

```
    result += i;
#ifdef DEBUG
    // 제대로 돌아가는 지 검사
    cout << i << ": 중간 결과 = " << result << endl;
#endif
  }
  // 결과 출력
  cout << "결과 = " << result << endl;
  return 0;
}
```

앞의 예의 경우, 맨 첫째 줄의 DEBUG이 정의되어 있지 않은 상태에서 컴파일하면, 중간 결과를 출력하는 부분이 컴파일되지 않아서 결국 최종 결과만 출력하고, 그렇지 않으면 중간 결과도 출력할 것이다.

#ifdef 과는 반대로 #ifndef 이라는 지시어는 특정 매크로가 정의되어 있지 않을 때에 대한 조건부 컴파일 지시어이다. 다음의 예는 SIZE라는 매크로가 정의되어 있지 않을 때, 정의하는 방법이다.

```
#ifndef SIZE
#define SIZE 100
#endif
```

그 외에 #if 와 #elif 지시어도 있다. 다음은 여러 가지 서로 다른 컴퓨터 시스템에서 컴파일되도록 하기 위해 매크로 정의를 사용하는 예제이다.

```
#if SYSTEM == 1
#include "ibm.h"
#elif SYSTEM == 2
#include "vax.h"
#elif SYSTEM == 3
#include "mac.h"
#else
#include "ibm.h"
#endif
```

SYSTEM 이라는 매크로가 어떻게 정의되어 있느냐에 따라 거기에 맞는 컴퓨터 시스템의 규격 데이터를 가지는 헤더 파일을 삽입하는 것이다.

조건부 컴파일이 특히 많이 사용되는 예는 C 언어의 헤더 파일에서이다. 헤더 파일은 근본적으로, 하나의 프로그램을 여러 개의 파일로 나누어 개발할 때 함수나 전역 변수에 대한 선언을 공유하거나, C/C++ 언어에서 기본적으로 지원하는 객체나 라이브러리에 대한 선언과 정의를 프로그래머의 프로그램에서 용이하게 사용하기 위해 도입되었다.

문제는 여러 소스 파일들에서 같은 헤더 파일을 여러 번 삽입하거나, 여러 개의 헤더 파일들에서 다른 헤더 파일을 호출하는 경우도 있을 수 있다는 것이다. 이런 경우, 이미 선언된 전역 변수나 함수 등의 정의가 다시 중복되어 선언되거나 컴파일되는 일이 발생한다. 그렇게 되면, 컴파일러는 이미 선언되거나 정의된 것을 다시 선언하거나 정의한다고 불평하게 된다. 예를 들어 사용자의

프로그램에서 stdio.h 를 삽입하고 그 다음 행에 bool.h 를 삽입했는 데, 바로 그 bool.h 내에서 stdio.h 를 다시 삽입하는 경우가 그러한 경우일 것이다. 이러한 경우 stdio.h 에서 아무런 방어 장치를 하지 않는다면, stdio.h 내의 함수 정의나 전역 변수들이 다시 선언되고 정의되어 에러가 발생한다.

이를 막기 위해 stdio.h 파일의 처음과 끝 부분에 조건부 컴파일 지시어인 #ifndef, #define, 그리고 #endif 를 다음과 같이 사용할 수 있다.

```
#ifndef __STDIO_H__
#define __STDIO_H__

/* stdio.h 의 내용 */

#endif
```

위와 같이 헤더 파일의 실제 내용을 조건부 컴파일 지시어로 둘러싸 버리면 여러번 삽입이 되어도 처음 한번만 컴파일이 되는 것이다.

## 제 5 절 #을 이용한 전달 인자의 문자열화와 ##를 이용한 토큰의 결합

매크로를 사용하는 경우 유용한 두가지 팁을 설명하겠다.

우선, #의 경우를 설명하기 위해 다음의 코드를 보자.

```
#define PRINT_2X(x) cout << "2 * x = " << ((x)+(x)) << endl
#include <iostream>
using namespace std;
void main(void)
{
  int value=0;
  cin >> value;
  PRINT_2X(value);
  PRINT_2X(10);
}
```

위의 프로그램을 컴파일하고 10을 입력하면 “2 * x = 20” 이 두 번 출력된다.

여기서 중간의 x 대신 다음과 같이 #x를 넣어, 맨 처음 줄을 다음과 같이 바꾸어 보자. #x 로 정의하면 그 값이 아니라 x 자체가 문자열로 들어간다.

```
#define PRINT_2X(x) cout << "2 * " #x " = " << ((x)+(x)) << endl
```

컴파일해서 실행해 보면 결과는 다음과 같다.

```
10
2 * value = 20
2 * 10 = 20
```

우리는 문자열이 화이트 스페이스로만 떨어져 있으면 실제 컴파일될 때에는 하나로 붙는다는 사실을 알고 있다.

```
// 아래의 정의들은 동일함
char hello[100] = "Hello World";
char hello[100] = "Hello " "World";
char hello[100] = "H" "e" "l" "l" "o" " " "W" "o" "r" "l" "d";
```

앞의 경우처럼 10을 입력했을 때, PRINT_2X(value) 문에 대해 "2 * " #x " = "는 "2 * " "value" " = "로 바뀌고, 최종적으로 "2 * value = "이 되는 것이다. 또한 그 다음의 PRINT_2X(10) 문장은 "2 * " "10" " = "로 바뀌고, 최종적으로 "2 * 10 = "이 된다.

두번째로 ##를 사용하면 두 개의 토큰이 결합된다. 그 결합이란 게, 말 그대로 단순무식하게 그냥 붙는 것이다.

이를 이해하기 위해서는 다음의 프로그램 예를 보자.

```
#define CONCAT(a,b) a ## b
#include <iostream>
using namespace std;
void main(void)
{
  int arr[2] = {100,200};
  cout << CONCAT(2,4) << endl;
  cout << CONCAT("Hello ","World") << endl;
  cout << CONCAT(arr,[0]) << endl;
}
```

이 프로그램의 실행 결과는 다음과 같다.

```
24
Hello World
100
```