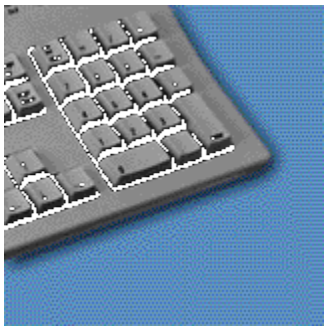


3. 전처리 명령어(preprocessing commands)

<장도비라>
C++의 고급 주제들을 더 살펴보기 전에, 이 장에서는 전처리 명령어에 대해서 살펴본다. 전처리 명령문은 컴파일 전에 처리되는 명령문으로 컴파일러에게 행동을 지시한다고 해서, 컴파일러 지시자(compiler directive)라고도 한다. 이 장에서는 MFC의 코드 자동화를 위해서 사용되는 전처리 명령문의 동작을 이해하기 위해, C/C++에서 사용하는 일반적인 전처리 명령문들을 살펴본다. 또한 MFC가 자동으로 생성한 파일에 포함된 THIS_FILE과 DEBUG_NEW의 원리를 이해한다.
</장도비라>

MFC 코드는 많은 수의 전처리 명령어를 사용한다. 예를 들어 MFC의 다큐먼트(document)나 뷰(view) 클래스 구현 파일의 윗부분에 정의된 아래의 코드를 보자.

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

__FILE__이 이미 현재의 파일 이름을 나타내는 전처리 명령어의 특별한 변수임에도 불구하고, 왜 그것을 THIS_FILE[]이란 이름으로 재정의 했을까? 또한 _DEBUG 빌드(build)에서 new를 DEBUG_NEW로 정의한 것은 코드 생성에 어떤 역할을 하는가?

우리는 이 장에서 아래의 내용들을 살펴 볼 것이다.

- 전처리 명령어
- 매크로 상수와 매크로 함수
- 전처리 명령문에서 사용하는 연산자
- 미리 정의된 매크로 상수, 매크로 변수
- #pragma의 역할

- THIS_FILE을 정의한 이유
- DEBUG_NEW의 역할

전처리 명령어(preprocessing command)는 C의 키워드이지만, 좀 특이한 명령어이다. 전처리 명령어는 컴파일러가 C 소스를 기계어 코드(machine code)로 번역하는 처리(processing)전에 이루어지기 때문에 붙여진 이름이다.

전처리 명령어를 일반 처리 명령어와 구분하기 위해서 전처리 명령어는 모두 특수문자 **파운드(#)**로 시작한다. 아래는 사용 가능한 모든 전처리 명령어이다(모두 #으로 시작함을 알 수 있다).

```
① #include
② #define
③ #if
④ #ifdef
⑤ #ifndef
⑥ #elif
⑦ #else
⑧ #endif
⑨ #undef
⑩ #line
⑪ #error
⑫ #pragma
```

전처리 명령어 중 굵게 표시된 #include, #define, #ifndef 와 #endif는 대부분의 프로그램에서 빈번하게 사용되는 중요한 명령어이므로, 반드시 사용법을 확실하게 익혀두어야 하며, 다른 명령어는 많이 사용되지는 않는다.

리스트 중, ③~⑧번의 전처리 명령어들은 조건 컴파일 등을 위해 함께 사용하는 명령어들이다.

#include

<절도비라>

이 절에서는 소스 파일에 다른 소스를 끼워넣는 #include에 대해 살펴본다. #include가 컴파일 시간이 아닌 컴파일 전에 매크로 확장됨을 이해하고, 포함시킬 파일의 경로를 명시하지 않기 위해 비주얼 C++에서 경로를

미리 설정하는 방법을 알아본다.
</절도비라>

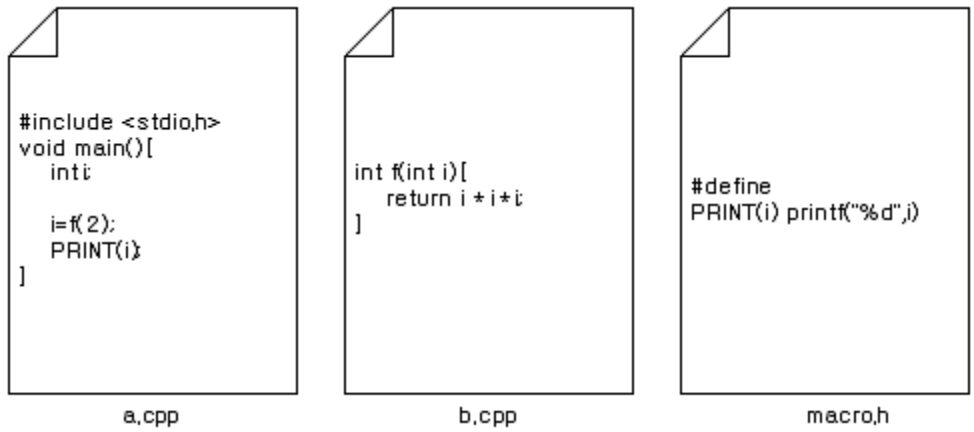
#include 명령문은 소스에 다른 소스파일을 '끼워넣기'하기 위해 사용한다.

소스 파일에 다른 소스 - 그것이 *.h 이든, *.cpp 이든 - 를 끼워넣는 과정을 생각해 보라. 이것은 분명 원래의 소스가 기계어 코드로 번역이 되기 전에 처리되어야 한다. 그래서 #include 문은 전처리 명령어이다. 실제로 #include 문은 컴파일 전에 이루어진다.

#include 문의 문법은 다음과 같다.

```
#include <header_name>
#include "header_name"
#include macro_identifier
```

아래 [그림 3.1]과 같이 3개의 소스 파일로 이루어진 프로그램을 생각해 보자.



[그림 3.1] 3개의 파일로 이루어진 프로젝트: 전체 프로그램을 구성하는 소스 파일이 2개 이상인 경우 이것을 결합하는 것이 필요하다. macro.h의 #define에 대해서는 아래의 절에서 자세히 설명한다. 지금은 단순히 PRINT(i)가 printf("%d",i)를 의미한다고 알아두자.

소스 파일이 2개 이상인 경우, 원하는 실행 파일을 만들기 위해서는 프로젝트 파일(project file)을 만들어야 한다. 하지만, 프로젝트 파일을 만들지 않고도 #include를 이용하여, 실행 파일을 만드는 것이 가능하다.

위의 예에서 a.cpp의 main()에서 b.cpp의 f()를 호출하고 있으며, 또한

macro.h의 PRINT() 매크로를 호출하고 있다. 함수나 매크로 함수 모두 호출하기 전에 선언되어야 하므로, a.cpp의 f()와 PRINT() 모두 컴파일 시간 에러를 유발한다.

해결 방법은 b.cpp의 소스와 macro.h의 소스를 그대로 a.cpp에 복사하여 사용하는 것이다. #include는 이러한 작업을 컴파일러가 코드 생성 전에 하도록 지시한다([그림 3.2]). a.cpp가 macro.h와 b.cpp를 자신의 상단 부분에 복사하기 위해 다음의 두 줄을 a.cpp의 소스에 추가한다.

```
#include "macro.h"
#include "b.cpp"
```

그러면 a.cpp의 소스는 다음과 같다.

```
#include <stdio.h>
#include "macro.h"
#include "b.cpp"
void main() {
    int i;
    i=f(2);
    PRINT(i);
}
```

실제로 위의 소스는 컴파일 전에 아래와 같이 **매크로 확장(macro expansion)**■된다.

<여기서 잠깐>

매크로 명령이 전처리 프로세서(preprocessing processor or macro processor)에 의해 처리되어 소스가 확장되는 것을 **매크로 확장**이라 한다.

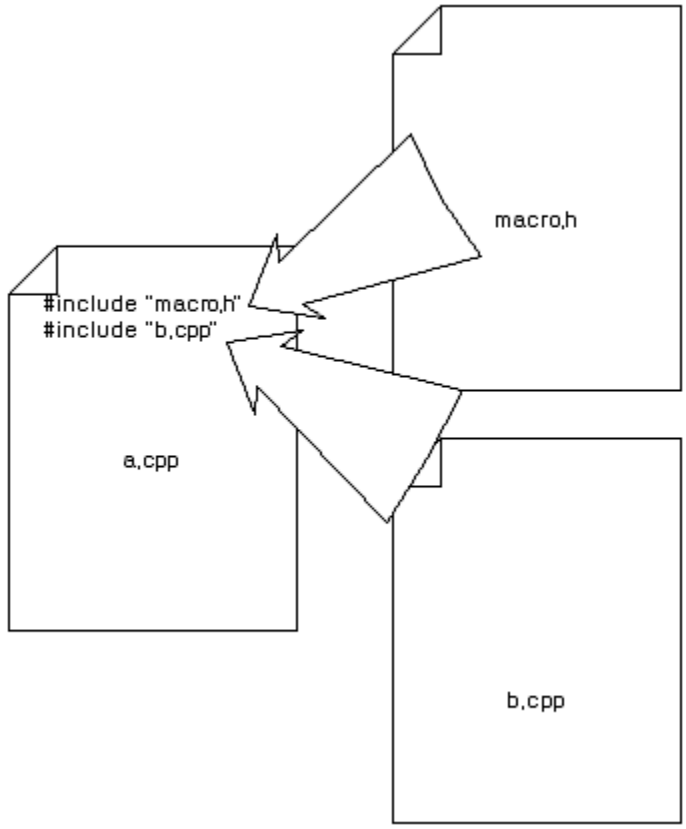
</여기서 잠깐>

```
#include <stdio.h>//이 문장은 일단 보류하자.
#define PRINT(i) printf("%d",i)
int f(int i) {
    return i*i*i;
}
void main() {
    int i;
    i=f(2);
```

```
    PRINT(i);
}
```

그리고 실제의 소스는 #define이 완전히 확장되어 컴파일 전의 완전한 소스는 아래와 같다.

```
#define <stdio.h>//실제로는 stdio.h가 이 부분에 확장될 것이다.
#define PRINT(i) printf("%d",i)
int f(int i) {
    return i*i*i;
}
void main() {
    int i;
    i=f(2);
    printf("%d",i);
}
```



[그림 3.2] #include의 역할: #include는 소스 파일의 중간에 다른 소스 파일을 포함(include)할 수 있다.

위의 예에서 편의 상 stdio.h의 확장은 소스에 고려하지 않았다. 독자들의 예상대로 특정한 소스 파일을 포함하는 전처리 명령은 #include 다음에 C의 문자열 표현을 사용하여 이중 인용 부호(double quotation mark: ") 사이에 파일의 경로를 명시하면 된다.

예를 들면, 현재 경로(current path)의 sub.h를 삽입하려면 아래의 예처럼 사용한다.

```
#include "sub.h"
```

만약 sub.h의 경로가 C:\VisualC++\MyProject\ 라면, 아래의 예처럼 사용할

수 있다.

```
#include "C:\VisualC++\MyProject\sub.h"
```

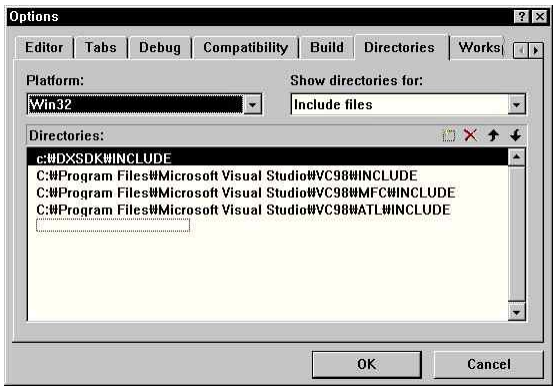
주의할 점은 #include가 명령문이 아니라, 전처리 명령문이므로, 위의 문장을 C의 Esc절차(Escape Sequence)를 사용하여, 아래의 예처럼 적어서는 안 된다는 것이다.

```
#include "C:\\VisualC++\\MyProject\\sub.h"
```

그러면, 아래의 문장은 무엇을 의미하는가?

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

파일 이름을 ""이 아니라 <와 >로 감싸는 것은 일반적으로 미리 정해진 경로(predefined path)의 헤더 파일를 포함하기 위해서 사용한다. 비주얼 C++ 6의 경우를 보자. 비주얼 C++은 메인 메뉴의 Tools->Options->Directories에서 미리 정해진 경로를 설정하는 대화상자를 제공한다. 대화 상자는 아래 [그림 3.3]과 같다.



[그림 3.3] Include Files: #include 의 미리 정해진 경로는 Include Files 에서 설정한다.

위의 예에서 #include의 디폴트 경로는 4개가 지정되어 있다. 제일 위의 것은 DirectX SDK를 위해 필자가 지정한 것이며, 나머지는 비주얼 C++이 인스

톨될 때 설정된 것이다.

그러므로 아래의 문장은 현재 경로에서 파일을 찾고 파일이 존재하지 않으면, 위의 네 경로에서 차례대로 stdio.h 파일을 찾는다. 그래도 파일이 없다면 에러가 발생한다□.

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

사실 비주얼 C++은 #include의 두 가지 문법을 구분하지 않는다. 하지만, 시스템 파일과 사용자 정의 헤더 파일을 구분하기 위해 두 가지 문법을 혼용하는 것은 코드를 읽기 좋게 만들므로, 시스템 헤더 파일인 경우 <와 >를, 사용자가 작성한 헤더 파일은 경우 “와 ”를 사용하는 것이 좋다.

위의 소스에서 발생할 수 있는 좀 더 미묘한 문제를 살펴보기 전에 #define에 대해 알아보자.

#define

<절도비라>
이 절에서는 매크로 상수와 매크로 함수를 정의하기 위해 사용하는 #define에 대해서 알아본다. 또한 매크로 상수의 이점과, 매크로 함수를 사용할 때의 주의점을 살펴보고, #include와 함께 사용되었을 때 발생할 수 있는 문제점을 이해한다.
</절도비라>

#define은 **매크로 상수와 함수를 정의(macro definition)**하기 위해서 사용한다. #define의 문법은 다음과 같다.

```
#define macro_identifier <token_sequence>
#define macro_identifier(<arg_list>) token_sequence
```

예를 들면 아래의 문장은 PI를 3.141592로 정의한 것이다.

```
#define PI 3.141592
```

그러므로 소스에서 PI는 컴파일 전에 3.141592로 치환된다. 즉, 소스의 다음과 같은 문장은

```
printf("%f\n",PI);
```

컴파일전에 아래의 문장처럼 치환되어 번역될 것이다.

```
printf("%f\n",3.141592);
```

이 때 PI는 상수 3.141592처럼 사용되었으므로 PI를 **매크로 상수(macro constant)**라 한다.

매크로 상수를 정의할 때 주의해야 하는 사항은 단어 사이의 공백에 관한 것이다. 예를 들면 아래의 문장은 A B를 C로 정의하는 것인가? A를 B C로 정의하는 것인가?

```
#define A B C
```

독자들이 문법을 주의 깊게 보았다면, 이것은 A를 B C로 정의한다는 것■

<여기서 잠깐>

명칭(identifier) - 변수 이름, 함수 이름 등 - 에는 공백이 올 수 없다.

</여기서 잠깐>

을 알았을 것이다. 즉 #define은 #define 이후의 첫 번째 단어(공백으로 구분된다)를 나머지 문자열로 정의한다.

아래의 소스는 #define을 이용하여 C언어를 전혀 다른 것처럼 나타낼 수 있음을 보여준다.

```
#include <stdio.h>

#define PI          3.141592
#define A           B, C
#define begin       {
#define end         }
#define procedure   void

int B=1,C=2;
```

```
procedure PrintPI()
begin
    printf("%f\n",PI);
end

void main()
begin
    PrintPI();
    printf("%d,%d\n",A);
end
```

결과는 다음과 같다.

```
3.141592
1,2
```

독자들은 C에서 블록(block)을 나타내기 위해 {와 }를 사용한다는 것을 알고 있을 것이다. Pascal 언어에서 블록은 begin과 end로 나타낸다. 또한 Pascal 언어에서 void 타입 함수는 **프로시저(procedure)**라고 한다.

이제 #include의 마지막 문법을 살펴 볼 준비가 되었다. 그것은 #define으로 정의된 매크로 상수를 #include에서 사용할 수 있다는 것이다. 아래의 문장을 고려해 보자.

```
#define files "c:\VisualC++\bin\macro.h"
#include files
```

위 문장은 아래의 문장과 동일하다.

```
#include "c:\VisualC++\bin\macro.h"
```

왜 매크로 상수를 사용하는가?

왜 매크로 상수를 사용하는가? 그것은 다음과 같은 몇 가지 이점 때문이다.

① 상수에 비해 메모리를 차지하지 않는다.

② 프로그램을 읽기 좋게 만든다.
③ 프로그램의 유지, 보수를 쉽게 만든다.

① #define PI 3.141592를 상수 표현식을 써서, 아래의 예처럼 사용하면 메모리를 4바이트 차지하게 된다.

```
const double PI=3.141592;
```

물론 단점도 존재한다. 디버깅 시에 매크로 상수 PI는 관찰할 수 있는 값이 아니다(모두 치환되어 있을 것이므로).

② 반지름 5인 원의 둘레를 계산하기 위해 아래처럼 사용한 문장을 생각해 보라.

```
float r=2*3.141592*5.0;
```

위 문장은 얼마나 코드를 읽기 어렵게 만드는가! 위의 경우 PI는 대부분의 사용자가 알고 있는 알려진 상수이기 때문에 정도가 덜하다.

어떤 프로그램이 자신의 상수 0.0012345678을 사용한다고 해보자. 이 상수는 샐러리 맨(salary man)의 월급에서 원천 징수되는 세금의 비율이라고 하자. 샐러리 맨의 월급 pay에서 세금을 계산하기 위해서

```
tax=pay*0.0012345678;
```

를 사용하는 것보다, 아래의 예처럼 사용하는 것이 확실히 읽기에 좋다.

```
#define RATIO    0.0012345678
tax=pay*RATIO;
```

③ 샐러리 맨의 월급에서 세금을 계산하는 위의 프로그램에서 RATIO는 얼마나 많이 사용될까? 예를 들어 200군데에서 RATIO를 사용한다고 하자. 만약 프로그래머가 매크로 상수를 사용하지 않았다면, 200군데에 0.0012345678이 나타날 것이다.

세금의 비율이 0.012345678로 100% 인상되었다고 하자. 프로그래머는 200군데의 소스를 수정해야 할 것이다. 모두 고쳐진다면 문제는 덜 심각하다. 실수로 1군데의 비율을 수정하지 못했다고 하자. 프로그래머는 알 수 없는 버그에

심각하게 고민해야 할 것이다.

하지만, 매크로 상수를 사용하면, 프로그래머가 수정해야 할 곳은 #define을 사용해서 상수를 정의하는 오직 한곳뿐이다!

매크로 함수(macro function)

```
#define macro_identifier(<arg_list>) token_sequence
```

위의 #define의 문법에서 두 번째 문법은 매크로 함수의 문법이다. 이것이 함수는 아니지만, 함수처럼 사용되기 때문에 이것을 매크로 함수라 한다. 하지만, 매크로 함수 역시 매크로 확장되는 것이지, 함수의 호출은 일어나지 않는다.

2개의 값 중 큰 값을 구하는 MAX(a,b)를 다음과 같이 정의할 수 있다.

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

위의 매크로 함수는 모든 MAX(a,b) 형태를 a>b?a:b 로 치환한다. 예를 들면, 아래의 문장을 고려해 보자.

```
printf("%d\n",MAX(2,3));
```

위 문장은 다음과 같이 치환되며, 결과는 3이 출력될 것이다.

```
printf("%d\n",2>3?2:3);
```

매크로 함수는 주의해서 사용하지 않으면, 심각한 문제가 발생한다. 아래의 예를 보자.

```
#include <stdio.h>

#define MUL_DEFINED
#define MUL(a,b) a*b

void main() {
    printf("%d\n",MUL(2+3,4));//20을 예상하는가?
}
```

결과는 다음과 같다.

```
14
```

결과가 14인 이유는 매크로 확장에 의해 명확해진다.

```
printf("%d\n",MUL(2+3,4));
```

위 문장의 매크로 확장은 2+3이 a에 해당하고, 4가 b에 해당하므로 다음과 같다.

```
printf("%d\n",2+3*4);
               a   b
```

연산자의 우선순위(priority)에 의해 결과는 14인 것이다. 매크로 함수를 정의할 때의 규칙은 다음과 같다.

"매크로 함수의 파라미터는 반드시 괄호로 묶어야 한다."

즉, MUL은 다음과 같이 정의되어야 한다.

```
#define MUL(a,b) (a)*(b)
```

또한, 매크로 함수간의 우선순위에 의해 문제가 발생할 수도 있으므로, 표현식 전체가 괄호로 묶이는 것이 바람직하다.

```
#define MUL(a,b) ((a)*(b))
```

위에서 정의했던, MAX() 역시 다음과 같이 정의되어야 한다.

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

위에서 살펴본 소스에서 아래의 문장에 주목했는가?

```
#define MUL_DEFINED
```

이 문장은 MUL_DEFINED라는 상수가 단지 정의되었다고 선언하는 것이다. 즉, 이 상수는 이미 정의되었으므로, 변수로도, 함수 이름으로도, 다른 매크로 상수로도 재정의(redefinition)할 수 없다. 그렇다고 MUL_DEFINED가 어떤 값을 가지는 것도 아니다.

이러한 #define의 역할에 대해, 이어지는 절(section)에서 살펴보도록 하자.

관례

매크로 함수와 상수는 관례(convention)상 모두 대문자를 사용한다. 위의 예들에서 모든 매크로에 대해 대문자를 사용했다. 이러한 규칙은 대부분의 C 프로그래머들이 지키는 관례이다. 독자들도 관례를 따르기를 바란다. 이러한 구분은 매크로와 일반 변수, 매크로 함수와 일반 함수를 구분하여 소스 코드를 읽기 좋게 한다.

미묘한 문제

이제 #include에서 언급했던, 미묘한 문제를 다룰 준비가 되었다. 다시 그 소스를 고려해 보자.

```
//a.cpp
#include <stdio.h>
#include "macro.h"
#include "b.cpp"
void main() {
    int i;
    i=f(2);
    PRINT(i);
}
//b.cpp
int f(int i) {
    return i*i*i;
}
//macro.h
#define PRINT(i) printf("%d",i)
```

위의 소스에서 b.cpp에서도 macro.h가 필요해서 b.cpp를 아래와 같이 수정

했다고 하자.

```
//b.cpp
#include "macro.h"
int f(int i) {
    PRINT(i);
    return i*i*i;
}
```

위의 프로그램이 컴파일될 것이라고 생각하는가? 그렇지 않다. 컴파일러는 매크로가 중복되었다는 에러 메시지를 출력할 것이다. a.cpp는 다음과 같이 매크로 확장된다(설명을 위해 #include만을 확장했다).

```
#include <stdio.h>

#define PRINT(i) printf("%d",i)
  //macro.h가 확장되었다.

#define PRINT(i) printf("%d",i)
  //b.cpp의 macro.h가 확장되었다. 이 문장에서 에러가 발생한다.
int f(int i) {
    PRINT(i);
    return i*i*i;
}
void main() {
    int i;
    i=f(2);
    PRINT(i);
}
```

위의 소스에서 보듯이, b.cpp의 macro.h가 확장될 때, 이것은 분명히 에러이다. PRINT()는 이미 정의된 매크로인 것이다.

그러므로, 헤더 파일을 만들 때는 이것이 프로젝트에 항상 1개만이 확장된다라는 보장이 되어야 한다. 이것을 구현하기 위해서, #if, #ifdef등을 사용한다.

#if와 defined 연산자

<절도비라>
이 절에서는 조건부 컴파일을 구현하기 위한 전처리 명령문인 #if와, #if에서 조건을 판단하기 위해서 사용하는 전처리 연산자 defined()의 문법을 익힌다. 아울러 #if와 defined()가 헤더 파일의 중복된 끼워넣기(include)를 방지하거나, 컴파일 조건에 따라 다른 코드를 생성하기 위해 사용될 수 있음을 살펴본다.
</절도비라>

#if의 문법은 다음과 같다. 이것은 블록(block)이 **섹션(section)**■으로 바뀐

<여기서 잠깐>
섹션은 블락으로 구분되지 않은 문장들을 말한다. #if의 조건이 끝나는 문장(즉 매크로 확장이 끝나는 문장)은 #elif, #else 혹은 #endif까지 이다.
</여기서 잠깐>

것을 제외하고는 if문의 문법과 같다.

```
#if <expression-1>
<section-1>
[#elif <expression-2>
<section-2>][...]
[#else
<section-3>]
#endif
```

독자들은 앞 장에서 전처리 명령문에서만 사용 가능한 연산자 #와 ##에 대해서 배웠을 것이다. 전처리 명령문에서 사용 가능한 또 하나의 연산자에

defined

가 있다. 이 연산자는 매크로 상수가 정의(definition)되어 있는지의 여부를 검사한다. 위의 예에서 macro.h는 다음과 같이 작성되는 것이 바람직하다.

```
#if !defined(PRINT)
#define PRINT
#define PRINT(i) printf("%d",i)
#endif
```

헤더 전체가 #if~#endif에 의해 감싸져 있다. 이것은 여러 곳의 소스에서 이 헤더 파일을 포함시키더라도 헤더가 한 번만 포함되는 것을 보장한다. 즉 정의가 이미 되어 있다는 것을 나타내기 위해 PRINT를 플래그(flag)로 사용한 것이다. 다시 확장된 a.cpp를 살펴보자.

//확장된 a.cpp

```
#include <stdio.h>

#if !defined(PRINT)//PRINT가 정의되어 있지 않으므로 아래의 두 문장은 확
                   //장된다.
#define PRINT
#define PRINT(i) printf("%d",i)
#endif

#if !defined(PRINT)//PRINT가 이미 위에서 정의되었으므로, 아래의 두문장은
                   //확장되지 않는다.
#define PRINT
#define PRINT(i) printf("%d",i)
#endif

int f(int i) {
    PRINT(i);
    return i*i*i;
}
void main() {
    int i;
    i=f(2);
    PRINT(i);
}
```

#ifdef와 #ifndef(if **n**ot **def**ined)를 사용하여 같은 일을 할 수 있다.

```
#if !defined(PRINT)
#define PRINT
...
#endif
```

위 문장은 아래의 문장과 동일하다.

```
#ifndef PRINT
#define PRINT
...
#endif
```

하지만, 두가지 이상의 복합 조건(compound condition)을 검사해야 하는 경우 #if를 사용하는 것이 효과적이다. 아래의 예에서는 PRINT가 정의되어 있고, DEBUG가 정의되어 있지 않으면, 매크로를 확장한다.

```
#if defined(PRINT) && !defined(DEBUG)
...
#endif
```

#if가 헤더의 중복을 피하기 위해서뿐만 아니라, 조건 컴파일을 위해서 사용되는 경우도 많다. 예를 들어, 독자들이 프로그램을 개발할 때만, 필요한 함수들에 대해서 생각해 보자. 이러한 함수들은 프로그램을 개발하거나, 디버깅할 때 필요에 의해서 만든 함수이므로, 프로그램이 완성되어 마지막 버전을 작성할 때는 포함되지 않는 것이 바람직하다(쓸데없는 함수의 포함은 프로그램의 덩치를 크게 한다). 이러한 함수를 f()라고 하면, 아래의 예는 #if를 사용하여 f()를 포함하고 제거시키는 기술을 보여준다.

```
#include <stdio.h>

#define DEBUG

#ifdef DEBUG
void f(int i) {
    printf("%d\n",i);
}
#endif

void main() {
    int i=2,j=3,k=4;
    k=i*j*k;
  #ifdef DEBUG
    f(k);
  #endif
    printf("%d,%d,%d\n",i,j,k);
}
```

위의 예에서 함수 f()의 정의와 main()에서 f()의 호출은 매크로 상수 DEBUG가 정의되어 있을 때에만 유효하다. 이 프로그램의 개발이 끝났다면, 이제 더 이상 f()는 필요 없으므로(유지 보수를 위해서 코드 자체는 남겨 두어야 한다), 코드에서 제거시키는 것이 바람직하다. 어떤 독자들은 이러한 문장을 주석으로 처리하여 구현할 지도 모르지만, 그것은 구식 방법이다.

위의 예에서 아래의 문장을

```
#define DEBUG
```

단순히 다음과 같이 처리하면 이러한 문제를 해결하게 되는 것이다.

```
//#define DEBUG
```

이제 #if와 #define의 규칙을 외워두자.

"#if와 #define은 헤더 파일이 중복으로 포함되는 것을 방지하기 위해서, 디버깅 모드와 릴리즈 모드(release mode)의 다른 버전의 프로그램을 작성하기 위해서 사용한다."

[주의] 비주얼 C++ 6.0에 포함된, MFC(Microsoft Foundation Class)의 자동 코드 생성기인 AppWizard가 자동으로 생성한 뷰 클래스(view class)의 헤더 파일을 보자. 위에서 언급한 2개의 기교를 모두 사용하고 있다. 아래의 예제는 프로젝트의 이름을 Test로 하여 AppWizard가 자동으로 생성한 코드의 TestView.h를 리스트한 것이다.

[예제 3.1] TestView.h

```
// TestDoc.h : interface of the CTestDoc class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_TESTDOC_H__C41CD2EC_B9DD_11D2_BC14_02608C6C
#define AFX_TESTDOC_H__C41CD2EC_B9DD_11D2_BC14_02608C6CA09B__INC

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
```

```
class CTestDoc : public CDocument
{
protected: // create from serialization only
          CTestDoc();
          DECLARE_DYNCREATE(CTestDoc)

// Attributes
public:

// Operations
public:

// Overrides
          // ClassWizard generated virtual function overrides
          //{{AFX_VIRTUAL(CTestDoc)
          public:
          virtual BOOL OnNewDocument();
          virtual void Serialize(CArchive& ar);
          //}}AFX_VIRTUAL

// Implementation
public:
          virtual ~CTestDoc();
#ifdef _DEBUG
          virtual void AssertValid() const;
          virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
          //{{AFX_MSG(CTestDoc)
                    // NOTE - the ClassWizard will add and remove member functions here.
                    //    DO NOT EDIT what you see in these blocks of generated code !
          //}}AFX_MSG
          DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the
previous line.

#endif // !defined(AFX_TESTDOC_H__C41CD2EC_B9DD_11D2_BC14_02608C6C
```

위의 [예제 3.1]에서 #if~#endif가 사용된 기교에 대해서 주목하기 바란다.

#undef, #line과 #error

<절도비라>

이 절에서는 중복된 매크로를 해제하기 위해 사용하는 #undef와, 컴파일 시점의 정보를 얻기 위해 줄 번호와 소스 파일 이름을 재정의할 수 있는 #line, 그리고 에러 메시지를 표현하기 위해서 사용하는 #error 등에 대해서 살펴본다.

</절도비라>

#undef는 이미 정의된 매크로를 해제한다. 예를 들어 아래의 문장은 에러가 아니다.

```
#define PRINT(i) printf("%d\n",i)
#undef PRINT
#define PRINT(i,j) printf("%d,%d\n",i,j)
```

위의 소스에서는 이미 정의된 PRINT를 재정의(redefinition)하기 위해 #undef를 사용하고 있다.

이러한 매크로의 재정의는 여러 사람이 프로그래밍하거나, 다른 사람이 작성한 코드의 일부를 사용할 때 충돌되는 매크로를 해결하기 위해서 사용한다.

특별한 매크로

ANSI가 정의한 미리 정의된 매크로(predefined macros)는 모두 6개이다. 그것은 아래와 같다.

```
__DATA__
__FILE__
__LINE__
__STDC__
__TIME__
__TIMESTAMP__
```

위 매크로들은 __FILE__과 __LINE__을 제외하고 모두 재정의할 수 없다. 비주얼 C++은 위의 표준외에도 추가적으로 미리 정의된 매크로들을 정의하고

있는데 MSDN에서

__cplusplus

로 검색해 보면 비주얼 C++이 지원하는 미리 정의된 매크로들을 살펴 볼 수 있다. __cplusplus는 코드가 C++모드로 컴파일되고 있음을 지시한다. 비주얼 C++에서만 지원되는 미리 정의된 매크로는 아래와 같다.

```
_CHAR_UNSIGNED
__cplusplus
_CPPRTTI
_CPPUNWIND
_DLL
_M_ALPHA
_M_IX86
_M_MPPC
_M_MRX000
_M_PPC
_MFC_VER
_MSC_EXTENSIONS
_MSC_VER
_MT
_WIN32
```

아래의 두 매크로는 그 중에서도 특별한데, 컴파일 과정 중에 문맥에 따라 값이 바뀐다. 그래서 매크로라기 보다는 컴파일러 변수라고 보는 것이 적절하다.

```
__FILE__
__LINE__
```

__FILE__과 __LINE__은 각각 현재 소스 파일의 이름과 줄 번호를 의미하는 매크로이다.

아래의 예를 참고하라(소스 파일을 work.cpp로 저장하였다).

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

```
void main() {
    int i=2,j=3,k=4;

    printf("%d,%d,%d\n",i,j,k);
    printf("%d\n",__LINE__);
    printf("%s\n",__FILE__);
}
```

출력 결과는 다음과 같다.

```
2,3,4
7
%PATH%WORK.CPP
```

__LINE__은 현재의 줄 번호를 십진 상수로 정의한다. 위의 예에서 결과가 7인 이유는 printf("%d\n",__LINE__); 가 work.cpp의 7번째 줄이기 때문이다.

__LINE__이 매크로로 간주되지만, 컴파일 시간에 값이 갱신된다는 점을 주의하라. 그러므로 __LINE__과 __FILE__은 컴파일러 변수로 보는 것이 바람직하다.

#line은 매크로 상수 __LINE__과 __FILE__을 재설정하기 위해 사용한다. 문법은 다음과 같다.

```
#line integer_constant ["filename"]
```

예를 들면, 아래의 문장은 #line이후부터 __LINE__을 10부터 카운트하며, __FILE__을 hello.cpp로 설정한다.

```
#line 10 "hello.cpp"
```

__FILE__과 __LINE__을 사용하는 아래의 예를 참고하라.

```
#include <stdio.h>

void main() {
#line 1 "main.cpp"
    int i=2,j=3,k=4;
```

```
    printf("%d,%d,%d\n",i,j,k);
    printf("%s, %d\n",__FILE__,__LINE__);
    printf("%d\n",__LINE__);
#line 3897
    printf("%s, %d\n",__FILE__,__LINE__);
}
```

출력 결과는 다음과 같다.

```
2,3,4
main.cpp, 4
5
main.cpp, 3897
```

#error는 컴파일 과정동안 에러 메시지를 출력하기 위해 사용한다. 아래의 예를 참고하라.

```
#include <stdio.h>

#define TYPE 1

void main() {
    int i=2,j=3;
  #if (TYPE!=0)
  #error You must define TYPE to 0
  #endif
    printf("%d,%d\n",i,j);
}
```

위의 소스는 컴파일되지 않는다. 컴파일 과정 중에 다음과 같은 컴파일 에러 메시지가 출력된다.

```
Error directive: You must define TYPE to 0 in function main()
```

운영체제나 환경에 의존적인 설정이 필요하

다면?

<절도비라>

이 절에서는 운영체제나 플랫폼에 종속적인 코드 생성에 필요한 #pragma의 역할에 대해서 이해한다. 또한 #pragma가 라이브러리를 지정하거나, 컴파일러의 트릭을 이용해서 TODO 리스트를 자동으로 생성하는 데에도 사용될 수 있음을 살펴본다.

</절도비라>

특정한 플랫폼(CPU와 운영체제)에 의존적이거나 각 컴파일러 개발사들의 독자적인 코드 생성을 돕기 위해 **#pragma**가 존재한다. 예를 들면 inline함수에서 inline함수를 호출하는 inline함수를 호출하는 경우, 모두가 인라인 확장될 것인가? 이 문제는 컴파일러 개발자들이 설정한 인라인 확장의 깊이(depth)에 의존한다. 비주얼 C++의 경우, 인라인 확장의 깊이는 다음과 같이 설정할 수 있다.

```
#pragma inline_depth
```

#pragma 뒤에 오는 어떤 것도 표준은 아니지만, 아래의 것들은 비주얼 C++에서 자주 사용되는 #pragma 지시어(directive)이므로 알아둘 필요가 있다.

```
#pragma comment
[#pragma pack]
#pragma message
#pragma once
#pragma warning
```

#pragma comment

#pragma comment의 가장 자주 사용되는 예는 표준 라이브러리외의 추가적인 라이브러리를 소스 코드에서 지정할 때이다. 표준 라이브러리라 함은 아래처럼 지정된 폴더에 있는 파일들을 말한다.

%Program Files%Microsoft Visual Studio\Vc98\Lib\

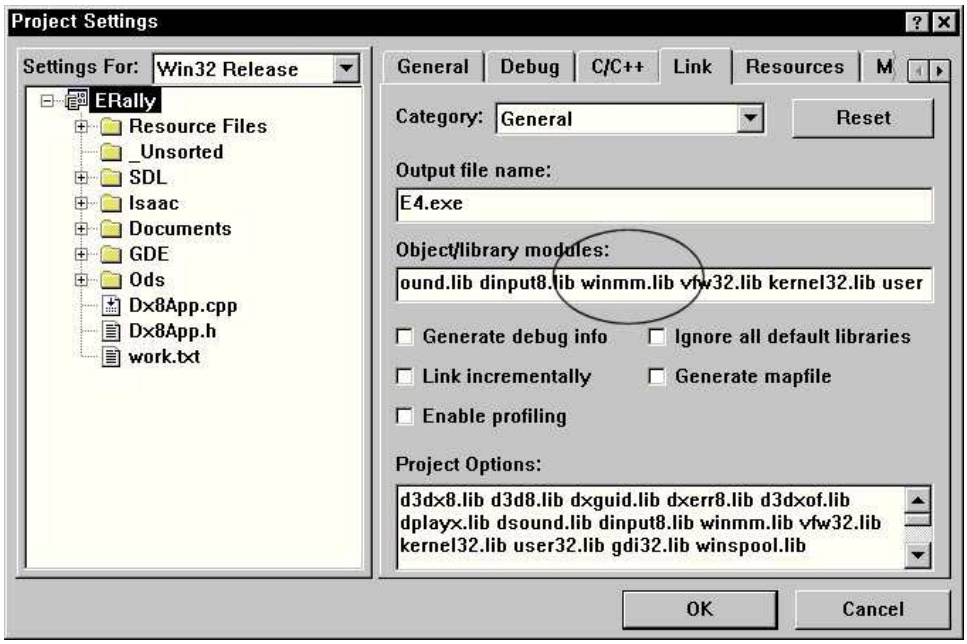
이러한 파일들에는 다음과 같은 것들이 있다.

```
LibC.lib
LibCD.lib
LibCMT.lib
LibCMTD.lib
LibCP.lib
LibCPD.lib
LibCPMT.lib
LibCPMTD.lib
```

위의 파일들은 C/C++ 표준 라이브러리이며, 파일이름을 구성하는 문자의 의미는 다음과 같다.

```
C: C용 라이브러리
CP: C++용 라이브러리
D: 디버거 버전
MT: 멀티 쓰레드(multi thread) 버전
```

예를 들면 LibCMTD.lib는 멀티 쓰레드용 C의 디버거 버전 라이브러리를 의미한다. 표준 라이브러리 이외의 추가적인 라이브러리를 링크 시간에 제대로 링크시키기 위해서는 링크할 라이브러리를 명시적으로 지정해 주어야 한다. 그것은 Project Settings의 Link탭에서 할 수 있다. 아래 [그림 3.4]를 보자.



[그림 3.4] 비표준 라이브러리의 지정: DirectX나 멀티미디어 라이브러리(winmm.lib)등의 비표준 라이브러리는 프로젝트 설정에서 명시적으로 지정해 주어야 한다.

위의 [그림 3.4]에서 윈도우용 멀티미디어 라이브러리를 지정한 것을 볼 수 있다. 이와 같은 비표준 라이브러리의 지정은 또한 #pragma comment를 사용해서도 할 수 있는데, 비표준 라이브러리를 포함하는 공개된 소스를 사용하는 경우, 프로젝트마다 라이브러리 설정을 해야 하는 번거로움이 줄며, 어떤 라이브러리를 사용하는 것이 문서화된다는 이점이 있다. 라이브러리 지정의 예는 아래와 같다.

```
#pragma comment(lib, "winmm.lib")
```

#pragma message

#pragma message는 컴파일 시간에 정보를 출력하기 위해 사용한다. 예를 들면 아래의 소스는 인텔 계열의 CPU를 사용하는 경우, 빌드(build) 창에 Pentium processor build를 출력한다.

```
#if _M_IX86 == 500
#pragma message( "Pentium processor build" )
#endif
```

#pragma message의 용도를 보여주는 아래의 예를 참고하자.

```
#include <stdio.h>

void main()
{
    printf( "hello\n" );
#pragma message( "between hello and world\n" )
    printf( "world\n" );
}//main()
```

위 프로그램을 컴파일하면 빌드 창에 다음과 같이 출력된다.

```
--------------Configuration: Console - Win32 Release--------------
Compiling...
main.cpp
between hello and world
Linking...

audit.exe - 0 error(s), 0 warning(s)
```

#pragma message는 일반적으로 매크로를 확인하여 정보를 출력하기 위해서 사용하지만, 다른 응용도 가능하다. 예를 들면, 긴 프로젝트에서 다음으로 기능 구현이 미루어진 부분을 표시하여, 매 빌드 때마다 빌드 창에 해야 할 일에 관한 정보를 출력하여 비주얼 C++의 에러 찾기 기능(F4)으로 찾아가도록 매크로를 정의할 수 있다. 이 매크로를 TODO()라 하자. 그러면 아래와 같이 매크로를 정의한다.

```
#define LINE1(x) #x
#define LINE(x) LINE1(x)
#define TODO(msg) message ( __FILE__ "(" LINE(__LINE__)  "): [TODO] "
msg )
```

이제 아래의 소스를 컴파일해 보자.

```
#include <stdio.h>

#define LINE1(x) #x
#define LINE(x) LINE1(x)
#define TODO(msg) message ( __FILE__ "(" LINE(__LINE__)  "): [TODO] "
msg )

void main()
{
#pragma TODO( "implement main()\n" )
    printf( "hello\n" );
    printf( "world\n" );
}//main()
```

그러면 빌드 창에 다음과 같이 출력된다.

```
------------------Configuration: Console - Win32 Release------------------
Compiling...
main.cpp
C:\[My Projects]\Console\main.cpp(9): [TODO] implement main()
Linking...

audit.exe - 0 error(s), 0 warning(s)
```

이제 F4키를 누르면 #pragma TODO()라고 표시된 부분으로 커서가 이동한다. 이렇게 TODO() 매크로를 이용하면 해야 할 일들을 자동으로 관리하는 것이 가능하다. 아래 [그림 3.5]를 보자.

[그림 3.5] #pragma TODO(): 소스에 해야할 일을 명시하면 F4로 해야할 일들의 목록을 탐색하는 것이 가능하다.

이것은 F4를 누르면 빌드나 파일에서 찾기(Find in Files) 창에서

파일이름(라인번호):

으로 표시된 부분을 검색하는 비주얼 C++의 기능을 이용한 트릭인데, 많은 프로그래머들이 위와 같은 트릭을 이용하고 있다.

#pragma once

#pragma once는 pragma가 명시된 파일이 컴파일 동안 오직 한번만 포함되는 것을 지시한다. 물론 #define으로 중복된 포함을 방지할 수 있지만, #pragma once가 명시되면 컴파일러가 파일 열기 자체를 시도하지 않으므로 컴파일 속도가 빨라진다. MFC가 자동으로 생성한 코드의 헤더 파일에는 모두 이 지시어가 포함되어 있다.

#pragma warning

#pragma warning의 가장 빈번한 사용은 특정한 경고(warning)를 끄기(disable) 위해서 사용한다. 아래의 문장은

```
#pragma warning( disable : 4507 34 )  // Disable warning messages
                                      //  4507 and 34.
```

4507과 34번 경고를 끈다. 경고를 끄는 것은 대부분의 경우 바람직하지 않다. 경고가 발생하는 경우는 이를 에러처럼 취급하고, 경고가 발생하지 않도록 코드를 작성해야 한다. 하지만, 부득이하게 경고가 발생하는 상황을 맞이해야 하는 경우가 있다. STL코드를 사용할 때 이러한 상황은 자주 발생한다. 이 때는 신중하게 판단해서 STL을 사용하는 자신의 코드에 #pragma warning()을 포함하여 경고를 끄도록 한다(예는 부록 CD에 포함된 bimap.h를 보라).

THIS_FILE을 정의한 이유

<절도비라>

이 절에서는 MFC가 자동으로 생성한 구현 파일의 첫 부분에 THIS_FILE을 정의한 이유를 알아보자. 또한 THIS_FILE이 단순하게 무시되어도 대부분의 소스에서는 문제가 없음을 살펴보도록 하자.

</절도비라>

__FILE__의 주된 사용 용도는 디버깅을 위해서이다. 에러가 발생했을 때 파일과 라인 정보를 출력하기 위해 __FILE__과 __LINE__을 사용한다. __FILE__이 정확하지 않은 유일한 경우는 전처리 명령문에 의해 포함된 파일에서 에러가 발생한 경우이다. 예를 들면 a.cpp에서 b.cpp를 #include로 포함했다고 하자. 그러면 a.cpp에는

```
#include "b.cpp"
```

와 같은 문장이 있을 것이고, 이 문장은 컴파일 전에 매크로 확장될 것이므로, b.cpp에서 사용한 __FILE__은 b.cpp를 나타내지 않고 a.cpp를 의미하게 된다. 이러한 경우 b.cpp에서 발생한 에러에 대한 정보는 정확하지 않을 뿐 아니라, a.cpp에 존재하지 않는 에러를 찾기 위해 프로그래머는 많은 시간을 들여야

할 것이다. 물론 __FILE__이 나타내는 이름을 재정의하기 위해 #line을 사용할 수 있지만, 라인 정보에 대해서는 수동의 갱신이 필요하기 때문에 유연하지 않다. 그래서 MFC는 모든 곳에서 __FILE__대신에 THIS_FILE을 사용한다.

그래서 MFC가 자동으로 생성한 각 cpp파일에는 디버깅 모드의 정적 변수로 THIS_FILE이 정의되어 있다. 하지만, 이 정의 부분을 삭제하는 것은 대부분의 경우에 별 이상이 없고, 실제 아무 상관이 없다. 왜냐하면 상위 클래스의 헤더 파일에 이미 THIS_FILE이 __FILE__로 매크로 정의되어 있기 때문인데, 독자들이 작성한 경우에 문제가 되는 유일한 경우는 독자들의 cpp파일이 다른 cpp파일을 포함하는 경우이다. 헤더 파일에서는 문제가 되지 않는데 왜냐하면 대부분의 헤더에서 정의한 매크로를 cpp가 사용하므로 cpp파일의 몇 번째 줄에서 에러가 발생했다는 정보는 에러를 발견하는데 충분하기 때문이다.

cpp파일에서 다른 cpp파일을 포함하는 경우는 언제 발생하는가?

그러면 cpp파일에서 헤더 파일이 아닌 다른 cpp파일을 포함해야 하는 경우는 언제 발생하는 것일까? 한 예로 DirectX 9.x의 수학 라이브러리를 들 수 있다.

DirectX는 C와 C++을 모두 지원하는 라이브러리이다. 사실 DirectX는 COM으로 구현되어 있으므로 COM을 지원하는 어떠한 언어도 사용할 수 있다. DirectX 6.x에서 소스와 함께 제공되던 D3DX 라이브러리가 C와 C++프로그래머를 모두 지원하기 위한 기교를 살펴보자.

예를 들면 벡터를 나타내는 D3DXVECTOR3은 (x,y,z)를 의미하는 세 개의 float멤버를 포함한다. C프로그래머를 위해서 이러한 벡터를 파라미터로 받는 여러 개의 수학 함수를 제공하는데, C++ 프로그래머를 위해서는 이러한 수학 연산을 연산자 오버로딩을 이용하여 제공하고 있다. 하지만, 연산자 오버로딩의 소스는 C로 컴파일되는 경우, 포함되지 않아야한다. 그래서 D3DX의 설계자들은 C++과 관련된 소스를 별도의 *.inc에 넣어두고 소스가 C++모드로 컴파일되는 경우에만 *.inc를 포함하도록 소스를 작성해 두었다. 이 경우, 연산자 오버로딩 함수에서 ASSERT()를 사용한 경우, *.inc에 해당하는 파일명으로 정보를 출력하는 것이 바람직하다. 이러한 소스는 헤더 정보가 아니므로 일반적으로 확장자를 INL 혹은 INC를 사용하여 작성한다. MFC의 경우 INL을 사용하는데 INcLude를 의미한다.

간단하게 소스를 구현해 보기 위해 먼저 ASSERT()를 정의해 보자. MFC 프로그래밍을 하는 경우 ASSERT()는 이미 정의되어 있으므로 그냥 사용하면 된다. ASSERT()는 디버그 모드로 컴파일되는 소스에서만 동작하는 조건을 검사하고 조건 검사가 실패하는 경우, 에러를 조기에 발견하기 위해 사용한다. 우리는 ASSERT() 조건 검사가 실패하면 조건 검사가 실패한 소스 파일과 라인 번호(line number) 그리고 사용자 메시지를 출력하는 메시지 박스로 구현할 것이다. 그리고 릴리즈 모드로 컴파일되는 소스에서는 ASSERT()가 동작하지 않도록 구성한다. 소스는 아래와 같다. 이 파일을 **03_MyAfx.h**라 하자.

[예제 3.2] 03_MyAfx.h

```
#if !defined(MYAFX_H)
#define MYAFX_H

#include <windows.h>

#define THIS_FILE   __FILE__
#if defined(_DEBUG)
#define ASSERT(f) \
    {\
        if ( !(f) )\
        {\
            char buffer[80];\
            sprintf( buffer, "%s(%d)\n", THIS_FILE, __LINE__ );\
            MessageBox( NULL, buffer, "ASSERT", MB_OK );\
        }\
    }
#else
#define ASSERT(f)
#endif // defined(_DEBUG)

#endif // !defined(MYAFX_H)
```

이제 CVector2를 정의해 보자. 이 구조체는 2차원 벡터를 나타내는데, C++ 모드로 컴파일되는 경우, 연산자 오버로딩 함수들을 포함하도록 소스를 구성할 수 있다. 소스는 아래와 같은 형태가 될 것이다.

```
static char _szAfxVectorInl[] = "03_VectorCpp.inl"; // (1)
```

```
#undef THIS_FILE
#define THIS_FILE _szAfxVectorInl

struct CVector2
{
    float   _x;
    float   _y;

#if defined(__cplusplus) // (2)
#include "03_VectorCpp.inl"
#endif // defined(__cplusplus)
};

#undef THIS_FILE // (3)
#define THIS_FILE   __FILE__
```

C++모드의 경우, (1) 이미 정의된 THIS_FILE의 정의를 취소(undefine)하고, 포함된 파일이름을 의미하는 파일명으로 THIS_FILE을 정의한다. (2) struct CVector2의 내부에서는 C++모드인 경우 연산자 오버로딩의 정의를 담고 있는 아래 파일을 포함(include)한다.

```
03_VectorCpp.inl
```

(3) 마지막으로 포함이 끝나는 시점에서 THIS_FILE을 다시 원래의 정의로 되돌린다.

이제 독자들은 C++모드가 확실하다면, 자유롭게 연산자 오버로딩 함수들을 사용할 수 있을 것이다. 아래의 예는 C와 C++의 혼합 모드 프로그래밍에서 THIS_FILE이 어떻게 사용되는지를 보여준다. [예제 3.3]을 보자.

[예제 3.3] THIS_FILE의 적절한 사용

```
#include <stdio.h>
#include "03_MyAfx.h"

static char _szAfxVectorInl[] = "03_VectorCpp.inl";
#undef THIS_FILE
#define THIS_FILE _szAfxVectorInl

struct CVector2
```

```
{
    float   _x;
    float   _y;

#if defined(__cplusplus)
#include "03_VectorCpp.inl"
#endif // defined(__cplusplus)
};

#undef THIS_FILE
#define THIS_FILE   __FILE__

void Test(CVector2 v)
{
    printf( "(%g,%g)\n", v._x, v._y );
    ASSERT(0);
}

void main()
{
#if defined(__cplusplus)
    CVector2    v(0.f,0.f);
    CVector2    v2(1.f,1.f);
    CVector2    v3(2.f,2.f);

    v = v2 + v3;
#else
    CVector2    v;
    CVector2    v2;
    CVector2    v3;

    v._x = v._y = 0.f;
    v2._x = v2._y = 1.f;
    v3._x = v3._y = 2.f;

    v._x = v2._x + v3._x;
    v._y = v2._y + v3._y;
#endif // defined(__cplusplus)
    Test( v );
}//main()
```

03_VectorCpp.inl은 [예제 3.4]에 리스트하였다.

[예제 3.4] 03_VectorCpp.inl

```
#ifndef _VECTORCPP_INL
#define _VECTORCPP_INL

CVector2(float x, float y)
{
    _x = x; _y = y;
}

CVector2 operator+(CVector2 r)
{
    CVector2 temp(_x+r._x, _y+r._y);
    ASSERT(0);
    return temp;
}

#endif // _VECTORCPP_INL
```

정리하면, 독자들이 MFC가 자동으로 생성한 코드에서 THIS_FILE이란 정의를 만나면 단순하게 그것이 __FILE__을 의미한다고 생각하라. 보기 싫다면 그 코드는 삭제해도 상관없다.

DEBUG_NEW의 역할

<절도비라>

이 절에서는 MFC가 자동으로 생성한 구현 파일에 DEBUG_NEWW를 정의한 이유와 이것의 동작원리에 대해서 살펴본다. 또한, 메모리 릭을 자동으로 발견하기 위한 비주얼 C++의 함수를 이용하는 방법을 익힌다.

</절도비라>

사용자가 사용한 동적 메모리 할당(dynamic memory allocation)에서 **메모리 릭(memory leak)**이 발생했을 때, 그것을 알 수 없을까? 가능하다. 그것은 비주얼 C++이 제공하는 디버그용 런타임 메모리 할당 함수를 사용하는 것이다.

비주얼 C++은 **crtdbg.h** 헤더 파일에 C의 메모리 할당 함수에 대응하는 디

버깅용 함수와 C++의 new 연산자에 대응하는 오버로드된 새로운 연산자 함수 operator new()를 정의해 놓았다. crtdbg.h에서 다음과 같이 정의된 부분을 찾아 볼 수 있다.

```
#ifdef  _CRTDBG_MAP_ALLOC

#define malloc(s) _malloc_dbg(s, _NORMAL_BLOCK, __FILE__, __LINE__)
#define  calloc(c,  s)  _calloc_dbg(c,  s,  _NORMAL_BLOCK,  __FILE__,
__LINE__)
#define  realloc(p,  s)  _realloc_dbg(p,  s,  _NORMAL_BLOCK,  __FILE__,
__LINE__)
#define  _expand(p,  s)  _expand_dbg(p,  s,  _NORMAL_BLOCK,  __FILE__,
__LINE__)
#define free(p) _free_dbg(p, _NORMAL_BLOCK)
#define _msize(p) _msize_dbg(p, _NORMAL_BLOCK)

#endif  /* _CRTDBG_MAP_ALLOC */
...
#ifdef _CRTDBG_MAP_ALLOC

inline void* __cdecl operator new(unsigned int s)
        { return ::operator new(s, _NORMAL_BLOCK, __FILE__, __LINE__); }

#endif  /* _CRTDBG_MAP_ALLOC */
```

위의 소스에서 기존의 malloc()이나 new등이 디버그용 함수로 호출되기 위해서는 crtdbg.h를 포함하기전에 _CRTDBG_MAP_ALLOC를 정의하면 된다는 것을 알 수 있다. 디버그 함수들이 동작하도록 하는 소스는 다음과 같을 것이다.

```
#define _CRTDBG_MAP_ALLOC
#include <crtdbg.h>
```

하지만 위의 문장을 소스의 제일 처음에 포함하는 것만으로는 어떠한 정보도 얻을 수 없는데, 다음과 같이 디버그 함수를 제일 처음에 호출해 주면 메모리 릭에 대한 정보를 디버그 창에 출력하도록 할 수 있다.

```
_CrtSetDbgFlag( _CRTDBG_ALLOC_MEM_DF ¦ _CRTDBG_LEAK_CHECK_DF);
```

이러한 디버그 함수들이 디버그 빌드(debug build)에서만 동작하도록 소스를 자동화 할 수 있다. 먼저 다음과 같은 헤더 파일을 작성하고, 이 파일의 이름을 [예제 3.5]의 **03_Debug_New.h**라 하자.

[예제 3.5] 03_Debug_New.h

```
#if !defined(_03_Debug_New_Defined_)
#define _03_Debug_New_Defined_

#if defined(_DEBUG)
//#define _CRTDBG_MAP_ALLOC
#include <crtdbg.h>

static class CRT_MEMORY_CHECK
{
public:
    CRT_MEMORY_CHECK()
    {
        _CrtSetDbgFlag( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
    }//CRT_MEMORY_CHECK()
} _crt_memory_check;

void* operator new(size_t size, const char* filename, int line)
{
    printf( "operator new() called\n" );
    printf( "%s(%d)\n", filename, line );
    return ::operator new(size, _NORMAL_BLOCK, filename, line);
}//operator new()

#define THIS_FILE   __FILE__
#define DEBUG_NEW   new(THIS_FILE, __LINE__)

#endif // defined(_DEBUG)

#endif // !defined(_03_Debug_New_Defined_)
```

이 헤더 파일은 먼저 class CRT_MEMORY_CHECK 타입의 스태틱 객체를 만들어 객체의 생성자에서 아래의 함수를 호출한다.

_CrtSetDbgFlag()

이것은 main()이나 WinMain()보다도 이 함수가 호출되는 것을 보장한다. 그리고 DEBUG_NEW를 오버로딩된 operator new()를 호출하도록 정의한다. 이제 다음과 같은 소스를 작성해 보자. 아래의 소스는 03_Debug_New.h를 포함하고, 소스의 제일 첫 부분에 new를 DEBUG_NEW로 정의하는 문장을 가진다.

```
#include <stdio.h>
#include <stdlib.h>
#include "03_Debug_New.h"

#if defined(_DEBUG)
#define new DEBUG_NEW
#endif // defined(_DEBUG)

void main()
{
    int* p = new int(); // (1)
    *p = 1;
    printf( "%i\n", *p );
    //delete p; // (2)
}//main()
```

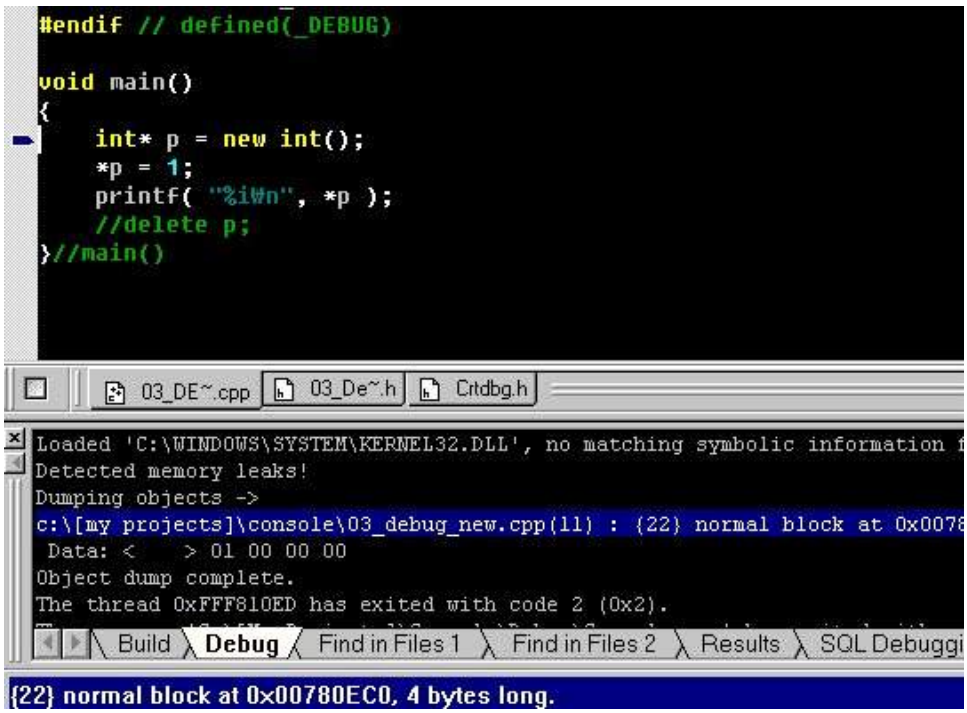
소스에서 주석 처리된 (2)부분을 주목하기 바란다. 의도는 동적으로 할당한 메모리를 해제하지 않은 에러를 발견하기 위해, 메모리 릭이 발생한 부분을 찾고자 하는 것이다. 그 부분은 (1)이 될 것이다. 위의 소스는 _DEBUG가 정의되어 있으면, 즉 디버그 빌드이면 new는 DEBUG_NEW로 치환된다. 그리고 DEBUG_NEW는 디버그용 메모리 할당 함수를 호출한다.

F5를 눌러서 디버그 런(debug run)한 다음 종료 후에 디버그 창을 보면 아래와 같은 정보가 출력된 것을 확인할 수 있다.

```
Loaded 'C:\WINDOWS\SYSTEM\KERNEL32.DLL', no matching symbolic
information found.
Detected memory leaks!
Dumping objects ->
c:\[my projects]\console\03_debug_new.cpp(11) : {22} normal block
at 0x00780EC0, 4 bytes long.
 Data: <    > 01 00 00 00
Object dump complete.
The thread 0xFFF810ED has exited with code 2 (0x2).
```

```
The program 'C:\[My Projects]\Console\Debug\Console.exe' has exited with
code 2 (0x2).
```

F4를 누르면 소스에서 메모리 릭이 발생한 부분을 가리킨다. 아래 [그림 3.6]을 보자.



[그림 3.6] 메모리 릭의 발견: 디버그용 메모리 함수를 사용하면 디버그 창에 메모리 릭의 정보를 출력해 준다.

릴리즈 빌드에서는 느린 디버그 함수들이 사용되는 것은 바람직하지 않다. 그래서 _DEBUG가 정의되지 않으면 어떠한 매크로 치환도 발생하지 않으므로 new는 원래의 new연산자를 호출하므로 빠르게 동작한다.

MFC에서는 지금까지 설명한 것과 같은 비슷한 방식으로 디버그 버전을 빌드한다. 그래서 MFC가 자동으로 생성한 각 CPP파일의 소스의 첫 부분에 DEBUG_NEW가 정의되어 있다.

독자들은 MFC의 소스를 보다가 설명하지 않은 전처리 명령문을 만나거나, 이해를 필요로 하는 매크로를 만나면 항상 **F12**를 눌러서 소스를 추적해 보기 바란다. 그리고 MSDN을 찾아보기 바란다. 이것은 후에 발견하기 힘들거나

이해하기 힘든 부분을 미연에 방지하는 좋은 습관이다.

요약

이 장에서 일반적인 전처리 명령문에 대해서 정리했다. defined 연산자를 이용하여 헤더 파일을 한번만 포함시키는 방법을 살펴보았으며, 플랫폼 독립적인 코드를 작성하기 위한 전처리 명령어들을 살펴보았다. 또한, MFC가 THIS_FILE을 각 .cpp파일마다 포함한 이유와 DEBUG_NEW를 사용하여 메모리 릭을 검출하는 원리를 공부했다.

- **전처리 명령어**는 컴파일러가 컴파일 처리를 하기전에 처리되는 명령어를 말한다.
- **매크로 상수**를 사용하는 이유는 프로그램을 읽기 좋게 만들고 유지/보수를 쉽게 하기 위해서이다.
- **매크로 함수**를 사용할 때 연산자 우선순위의 문제가 발생하지 않도록 파라미터를 괄호로 묶어주는 것이 관례이다.
- 헤더 파일이 중복되어 포함되는 것을 방지하기 위해 **#if와 defined 연산자**를 이용할 수 있다.
- 플랫폼에 독립적인 코드를 작성하기 위해서는 컴파일러 벤더가 제공하는 특별한 **매크로 변수**를 사용해야 한다.
- **#pragma comment**를 이용하면 소스 파일에 직접 링크할 라이브러리를 지정할 수 있다.
- MFC의 어플리케이션 위저드가 생성한 .cpp 파일에는 제대로 된 __FILE__을 참조하기 위해 **THIS_FILE**이 정의되어 있다.
- MFC의 어플리케이션 위저드가 생성한 .cpp파일에는 메모리 릭을 검출하기 위해 **DEBUG_NEW**가 정의되어 있다.

[문서의끝]