20. 전처리 명령어(preprocessing command)

- 컴파일러가 C 소스를 기계어 코드로 번역하는 것을 **처리(processing)**라고 합니다.
- 전처리 명령어는 이러한 '처리' 전에(pre) 실행되는 명령어입니다.
- 전처리 명령어를 일반 처리 명령어와 구분하기 위해서 전처리 명령어는 모두 특수문자 **파운드(#)**로 시작합니다.

- 아래는 사용 가능한 모든 전처리 명령어입니다.

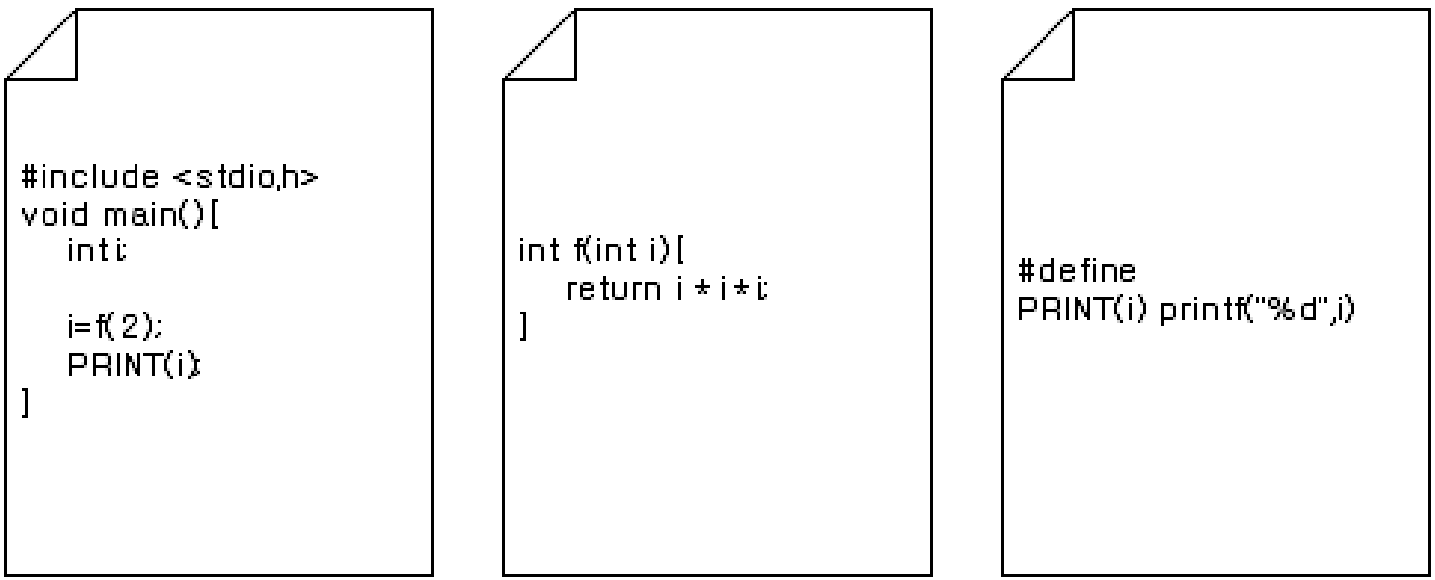
① #include
② #define
③ #if
④ #ifdef
⑤ #ifndef
⑥ #elif
⑦ #else
⑧ #endif
⑨ #undef
⑩ #line
⑪ #error
⑫ #pragma

#include

- #include 명령문은 소스에 다른 소스파일을 '끼워넣기'하기 위해 사용합니다.
- #include 문의 문법은 다음과 같습니다.

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

- 아래 그림과 같이 3개의 소스 파일로 이루어진 프로그램을 생각해 봅시다.



3개의 파일로 이루어진 프로젝트: 전체 프로그램을 구성하는 소스 파일이 하나 이상인 경우 이것들의 관련 정보를 설정하는 것이 필요합니다. macro.h의 #define은 PRINT(i)가 printf("%d",i)를 의미한다는 선언입니다.

• 위의 예에서 a.cpp의 main()에서 b.cpp의 f()를 호출하고 있으며, 또한 macro.h의 PRINT() 매크로를 호출하고 있습니다.
• 함수나 매크로 함수 모두 쓰기 전에 선언되어야 하므로, a.cpp의 f()와 PRINT() 모두 에러를 유발합니다.
• 해결 방법은 b.cpp의 소스와 macro.h의 소스를 그대로 a.cpp에 복사하여 사용하는 것입니다.
• #include는 이러한 작업을 컴파일러가 코드 생성 전에 하도록 지시합니다.

• 두 줄을 a.cpp의 소스에 추가합니다.

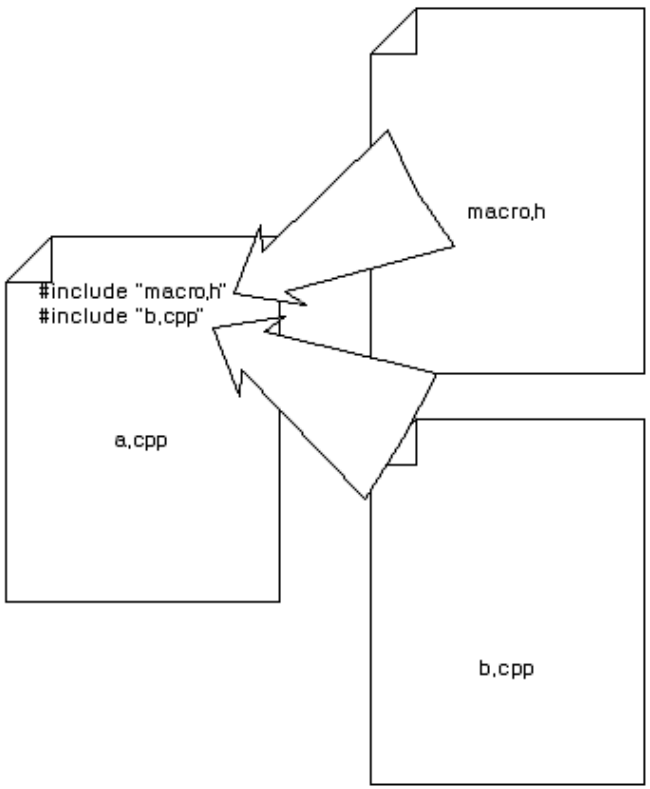
```
#include <stdio.h>
#include "macro.h"
#include "b.cpp"
void main() {
    int i;
    i=f(2);
    PRINT(i);
}
```

• 위의 소스는 컴파일 전에 아래와 같이 **매크로 확장(macro expansion)**됩니다.

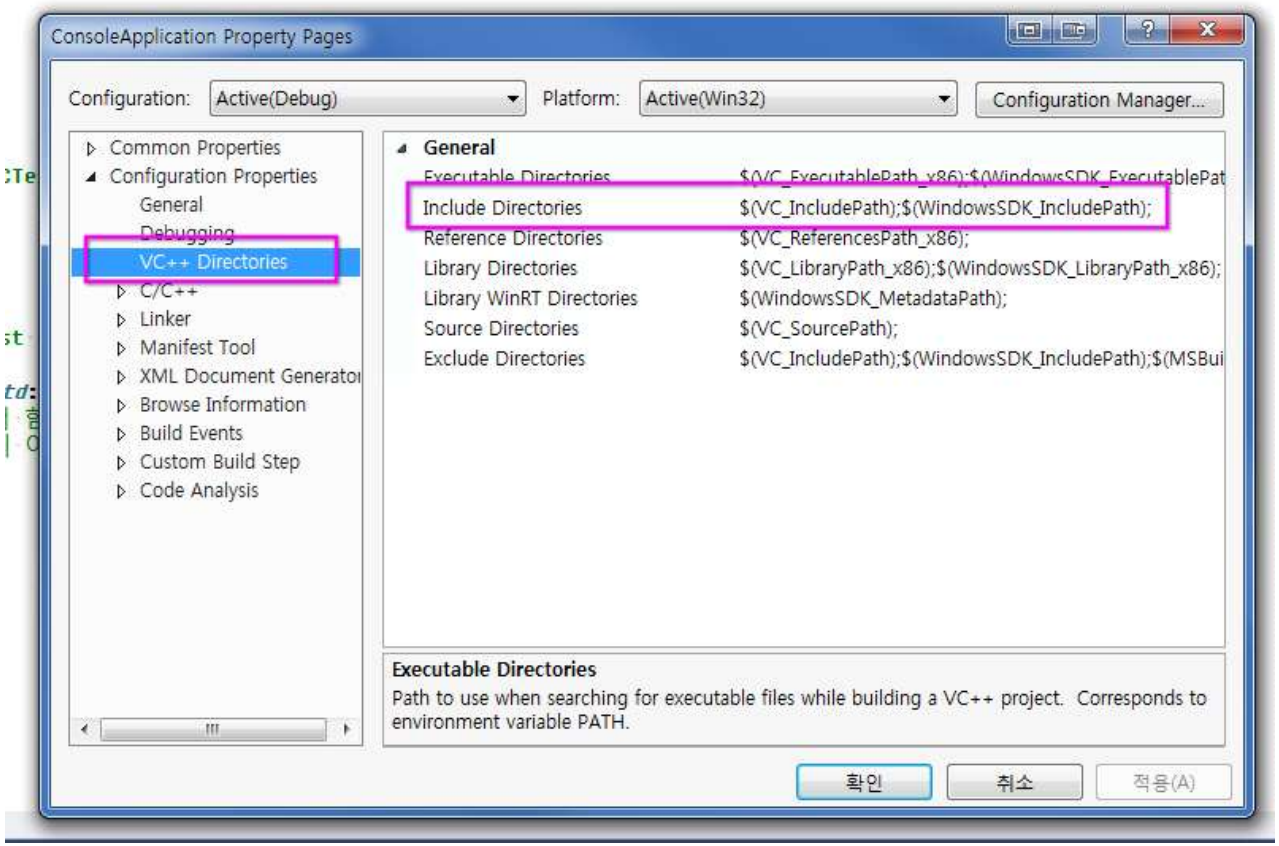
```
#include <stdio.h>//이 문장은 일단 확장을 보류합니다.
#define PRINT(i) printf("%d",i)
int f(int i) {
    return i*i*i;
}
void main() {
    int i;
    i=f(2);
    PRINT(i);
}
```

- #define이 완전히 적용되어 컴파일 전의 a.cpp의 완전한 소스는 아래와 같습니다.

```
#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);
}
```



#include의 역할: #include는 소스 파일의 중간에 다른 소스 파일을 포함(include)할 수 있습니다.



비주얼 스튜디오 2013의 디폴더 include 경로 설정: Visual Studio 2013에서는 프로젝트마다 기본 경로를 설정하는 것이 가능합니다.

#define

- #define은 **매크로 상수와 함수를 정의(macro definition)**하기 위해서 사용합니다.

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

- 상수 PI를 다음과 같이 정의할 수 있습니다.

```
#define PI 3.141592
```

- 위 문장은 PI를 3.141592로 정의한 것입니다. 이제 소스에서 PI라는 명칭은 컴파일 전에 3.141592로 치환됩니다.

• 소스에 다음과 같은 문장이 있다고 가정해 봅시다.

printf("%f₩n",**PI**);

• 위 문장은 전처리 단계에서 다음과 같이 확장됩니다.

printf("%f₩n",**3.141592**);

• 이 때 PI는 상수 3.141592처럼 사용되었으므로 PI를 **매크로 상수(macro constant)**라고 합니다.

- #define은 언어를 전혀 다른 것처럼 보이게 만들 수도 있습니다.

```
#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 같습니다.

```
3.141592
1,2
```

- 이제 #include의 마지막 문법을 살펴 볼 준비가 되었습니다.

```
#define files "c:₩borlandc₩bin₩macro.h"
#include files
```

- 위의 코드는 다음의 문장과 동일합니다.

```
#include "c:₩borlandc₩bin₩macro.h"
```

왜 매크로 상수를 사용하는가?

- 왜 매크로 상수를 사용하는 걸까요? 그것은 다음과 같은 몇 가지 이점 때문입니다.

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, 다음과 같이 정의할 수도 있습니다.

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

② 반지름 5인 원의 둘레를 계산하기 위해 아래의 코드를 사용했다고 가정해 봅시다.

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

③ 샐러리 맨의 월급에서 세금을 계산하는 위의 프로그램에서 RATIO는 얼마나 많이 사용될까요?

• 예를 들어 코드의 200군데에서 RATIO를 사용한다고 합시다. 만약 프로그래머가 매크로 상수를 사용하지 않았다면, 200군데에 0.00123456780이 나타날 것입니다.

• 세금의 비율이 0.012345678로 100% 인상되었다고 합시다. 프로그래머는 200군데의 소스를 수정해야 할 것입니다.

매크로 함수(macro function)

#define macro_identifier(<arg_list>) token_sequence

- 매크로 함수 역시 매크로 확장되는 것이지, 함수의 호출은 일어나지 않습니다.

- 두개의 값 중 큰 값을 구하는 MAX(a,b)를 다음과 같이 정의할 수 있습니다.

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

- 위의 매크로 함수는 모든 MAX(a,b) 형태를 a>b?a:b 로 치환합니다.

```
printf("%d₩n",MAX(2,3));
```

- 위의 문장은 다음과 같이 치환됩니다.

```
printf("%d₩n",2>3?2:3);
```

- 결과는 3이 출력될 것입니다.

• 주의해서 사용하지 않으면, 심각한 문제가 발생할 수도 있습니다.

```
#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 같습니다.

14

• 대부분은 아마도 결과로 20을 예상했을 것입니다. 결과가 왜 14인지는 매크로 확장의 과정을 이해하면 명확해 집니다.

• 매크로 함수를 정의할 때의 규칙은 다음과 같습니다.

"**매크로 함수의 파라미터는 반드시 괄호로 묶여야 합니다.**"

• MUL은 다음과 같이 정의되어야 합니다.

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

• 매크로 함수간의 우선 순위에 문제가 발생할 수도 있으므로, 표현식 전체를 괄호로 묶는 것이 바람직합니다.

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

- 우리가 살펴본 소스에서 아래의 문장이 존재합니다.

```
#define  MUL_DEFINED
```

- 이 문장은 MUL_DEFINED라는 상수가 단지 정의되었다고 선언하는 것입니다.
- 즉, 이 상수는 이미 정의되었으므로, 변수로도, 함수 이름으로도, 다른 매크로 상수로도 재정의(redefinition)할 수 없습니다.

관례

- 매크로 함수와 상수는 관례(convention)상 모두 대문자를 사용합니다.
- 이러한 규칙은 대부분의 C 프로그래머들이 지키는 관례입니다.
- 이러한 구분은 매크로와 일반 변수, 매크로 함수와 일반 함수를 구분하여 소스 코드를 읽기 좋게 합니다.

미묘한, 하지만 중요한 문제

- 이제 #include를 설명하면서 언급했던, 미묘한 문제를 다룰 준비가 되었습니다.

```
//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 수정했다고 합시다.

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

• 위의 프로그램이 컴파일될 것이라고 예상하지만 그렇지 않습니다.
• 컴파일러는 매크로가 중복되었다는 에러 메시지를 출력할 것입니다.

- a.cpp는 다음과 같이 매크로 확장됩니다(설명을 위해 #include만을 확장했습니다).

```
#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);
}
```

#if와 defined 연산자

• #if의 문법은 다음과 같습니다. 이것은 블록(block)이 **섹션(section)**으로 바뀐 것을 제외하고는 if문과 거의 유사합니다.

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

defined

- 이 연산자는 매크로 상수가 정의(definition)되어 있는지의 여부를 검사합니다.

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

//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);
}
```

• 2가지 이상의 복합 조건(compound condition)을 검사해야 하는 경우 #if를 사용하는 것이 효과적입니다.
• 아래의 예에서는 PRINT가 정의되어 있고, DEBUG가 정의되어 있지 않으면, 매크로를 확장합니다.

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

- 아래의 예는 #if를 사용하여 f()를 포함하고 제거시키는 기교를 보여줍니다.

```
#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);
}
```

#undef, #line, #error 와 #pragma

- #undef는 이미 정의된 매크로를 해제합니다.

```
#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 있습니다.

미리 정의된 매크로(predefined macros)

• 컴파일러 제조사는 컴파일러 에러 메시지 출력 등을 위해 몇 개의 매크로를 미리 정의하여 두었습니다.

```
__cplusplus
__FILE__
__LINE__
```

```
#ifdef __cplusplus
extern "C" {
#endif
void f();
int g(int);
void h(int,int);
#ifdef __cplusplus
}
#endif
```

• __FILE__과 __LINE__은 각각 현재 소스 파일의 이름과 줄 번호를 출력하는 매크로 상수입니다.

이 매크로는 한 번 정의되어 계속해서 같은 값으로 사용되는 일반적인 매크로 상수와는 다릅니다. 이 값은 컴파일 시간에 컴파일러에 의해서 설정되는 값을 가지는 컴파일러 변수이지만, 매크로에서 사용하므로 매크로라고 합니다.

```
#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 같습니다.

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

• __LINE__은 현재의 줄 번호를 십진로 정의합니다. 위의 예에서 결과가 7인 이유는 printf("%d₩n",__LINE__); 가 work.cpp의 7번째 줄이기 때문입니다.
• __LINE__이 비록 값이 갱신되기는 하지만, 이것은 변수가 아니라, 사용자가 재 정의할 수 없는 매크로 상수입니다.
• #line은 매크로 상수 __LINE__과 __FILE__을 재설정하기 위해 사용합니다. 문법은 다음과 같습니다.

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

- #error는 컴파일 과정동안 에러 메시지를 출력하기 위해 사용합니다.

```
#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);
}
```

- 우리가 작성한 코드가 오로지 C++모드에서만 컴파일되기를 원한다면?

```
#ifndef __cplusplus
#error You must compile this in C++ mode
#endif
```

운영체제나 환경에 의존적인 설정이 필요하다면?

• 만약 컴파일러 설계자에 의해 설정되어야 하는 값들이 표준 매크로로 나타내기 불가능한 것이라면, 어떻게 이러한 것을 처리할 수 있을까요?

• C++ 표준 위원회는 이러한 값들을 위해 **#pragma**를 추가하였습니다.

실습문제

1. #include "a.h"와 #include <a.h>를 구분하여 설명하세요.

2. 볼런드 사의 컴파일러는 자사의 컴파일러에 미리 정의된 매크로인 __BCPLUSPLUS__를 제공한다. 마이크로소프트 사의 컴파일러는 비주얼 C++에서 어떤 유일한 매크로를 정의하였는지 파악하고 설명하세요

3. 전처리 명령어에 #이 명시되지 않으면, 전처리 처리기(preprocessor)는 어떤 추가적인 작업이 필요할까요?

4. 전처리 명령문에서만 사용할 수 있는 연산자 #, ##과 defined에 대해서 설명하세요.

5. 사용자가 외부 라이브러리를 코드에 명시하기 위한 방법이 있을까요?(힌트: #pragma).

@