

8장 전처리기

박 종 혁 교수

UCS Lab

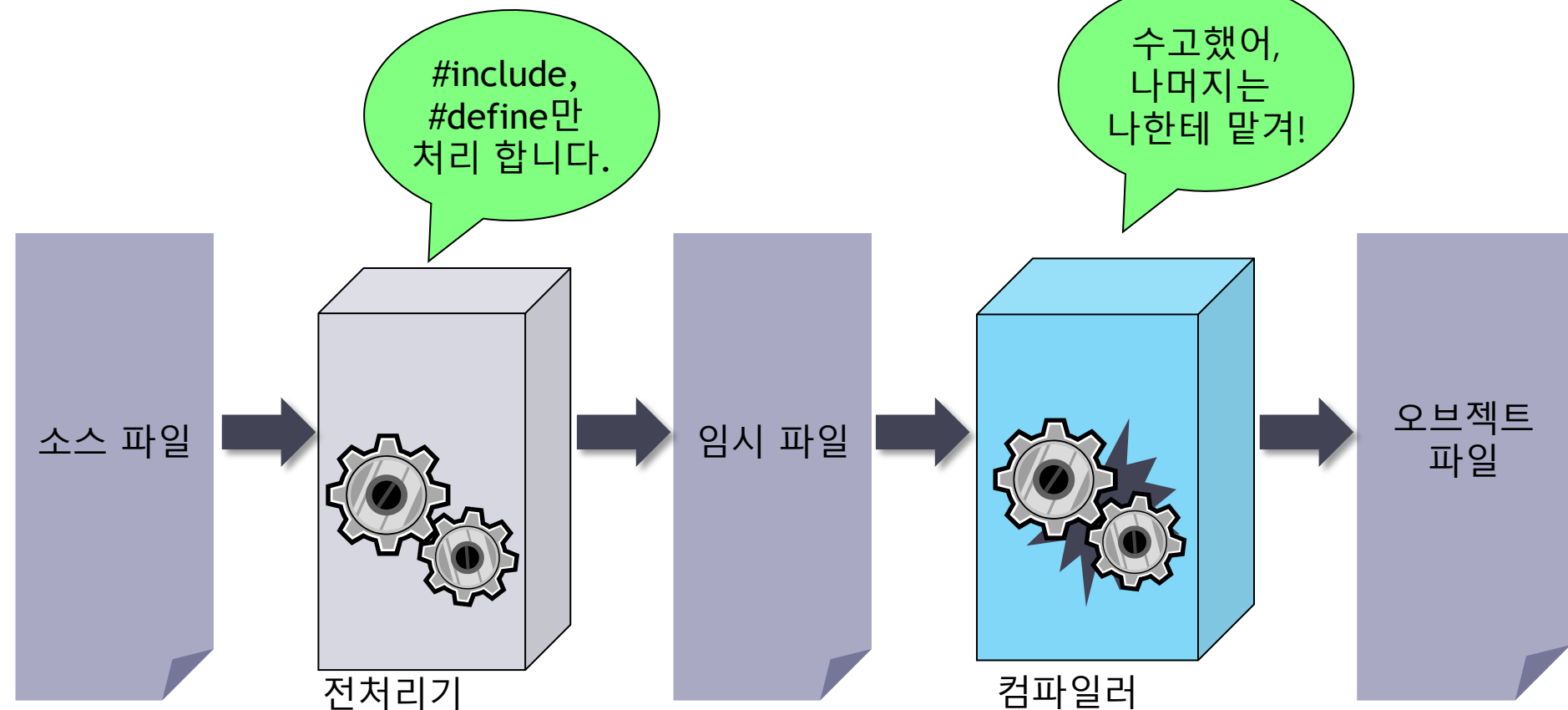
Tel: 970-6702

Email: jhpark1@seoultech.ac.kr

전처리기

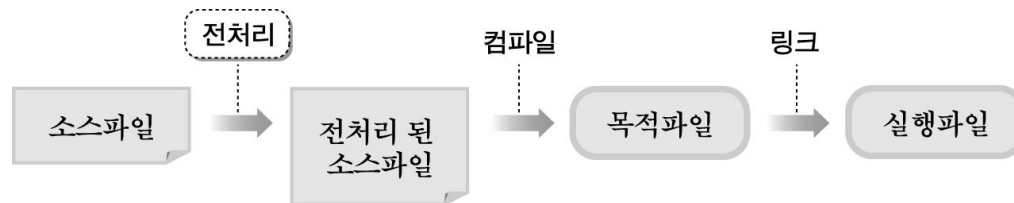
- C 언어는 전처리기를 사용하여 그 능력과 표기법을 확장함
- #으로 시작하는 행을 전처리 지시자라고 함
 - #include ...
 - #define ...
 - #if ...
 - #error ...
 - ...
- ANSI C에서 #은 여백 문자 다음에 올 수 있지만, 전통적인 C에서는 첫 번째 열에 #이 와야만 함
- 전처리 지시자가 영향을 미치는 범위
 - 그 파일에서 전처리 지시자가 있는 위치에서 시작여 그 파일의 끝까지
 - 다른 지시자에 의해서 그 지시자의 효력이 없어질 때까지

- 전처리기 (preprocessor)는 컴파일하기에 앞서서 소스 파일을 처리하는 컴파일러의 한 부분



전처리기

- 컴파일러는 목적파일을 만들기 전에 전처리라고 하는 특별한 작업을 먼저 수행한다.



- 전처리 과정에서 소스파일에 다른 파일의 텍스트를 포함시키거나 일부 문장을 다른 문장으로 바꾸는 작업 등을 수행한다.
- 전처리 명령어는 '#'기호로 시작한다.
- 전처리가 끝난 파일 역시 소스파일과 마찬가지로 텍스트 파일이다.

전처리기의 요약

지시어	의미
#define	매크로 정의
#include	파일 포함
#undef	매크로 정의 해제
#if	조건이 참일 경우
#else	조건이 거짓일 경우
#endif	조건 처리 문장 종료
#ifdef	매크로가 정의되어 있는 경우
#ifndef	매크로가 정의되어 있지 않은 경우
#line	행번호 출력
#pragma	시스템에 따라 의미가 다름

#include

- 지정된 파일을 읽어 드림
 #include "filename"
 #include <filename>
- #include에 명시되는 파일의 내용은 제한이 없음
- 또한 그 파일은 전처리기에 의해 다시 확장되어야 하는 또 다른 전처리 지시자를 포함할 수도 있음

#include

#include "filename"

- 이 행은 filename 파일의 사본으로 대체됨
- filename 파일은 먼저 현재 디렉토리에서 검색하고, 거기에 없다면 시스템이 정의한 디렉토리에서 검색함

#include <filename>

- 이것은 시스템이 정의한 디렉토리에서만 검색함
- UNIX 시스템에서 stdio.h와 stdlib.h 같은 표준 헤더 파일은 /usr/include에 있음
- 일반적으로, 표준 헤더 파일이 저장된 장소는 시스템에 따라 다름

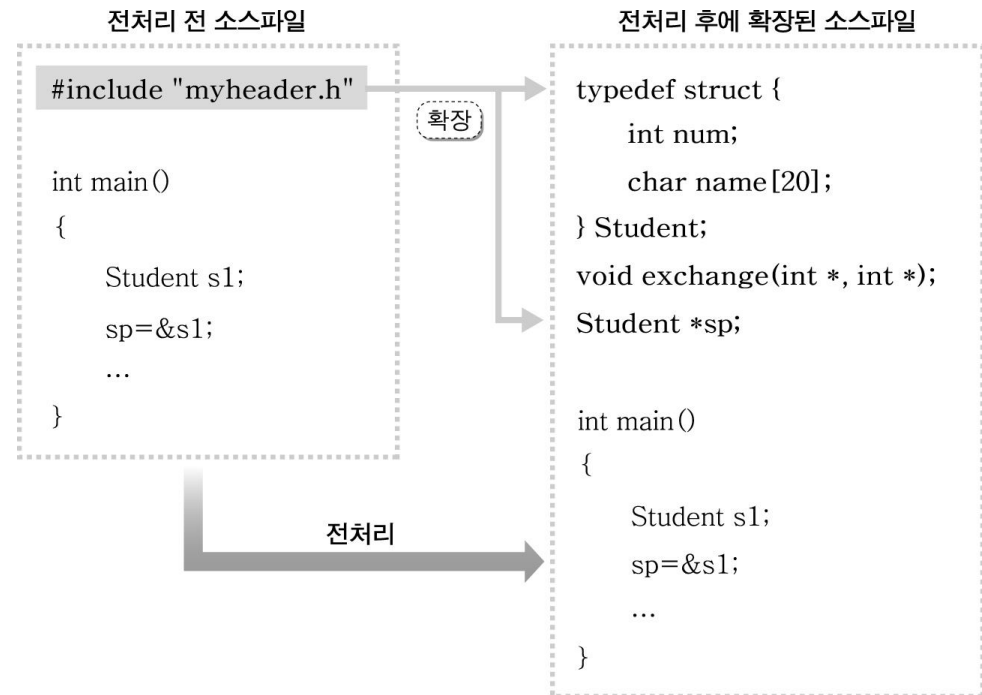
#include

- 헤더파일을 사용하면 프로그램을 깔끔하고 편리하게 작성할 수 있다.
 - 여러 프로그램에서 공통적으로 사용하는 구조체나 함수 또는 외부변의 선언을 헤더파일로 만들어 놓고 각 프로그램에서 간단히 포함하여 사용한다.

```
typedef struct {           // 구조체의 형 선언
    int num;
    char name[20];
} Student;

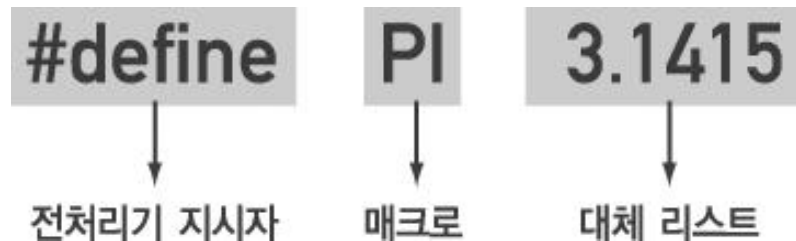
void exchange(int *, int *); // 교환 함수의 선언
Student *sp;                // 외부변수의 선언
```

⇒ **myheader.h**로 작성하여 현재의 작업 디렉토리에 저장해 둔다.



전처리기에 의한 매크로 처리

- **#define**으로 시작하는 전처리기 지시자
 - 컴파일러에 의해 처리되는 것이 아니다.
 - 전처리기에게 단순 치환 작업을 요청할 때 사용되는 지시자



#define

- 두 가지 형식

```
#define identifier token_stringopt
```

// 단순 매크로

```
#define identifier(id_1, ..., id_n) token_stringopt
```

//함수 매크로

- 정의가 길어질 경우에 현재 행의 끝에 역슬래시
\를 삽입하면, 다음 행에 연결해서 계속 쓸 수 있음

단순 매크로

- 단순 매크로(macro): 숫자 상수를 기호 상수로 만든 것
- (예)

```
#define MAX_SIZE 100  
#define PI 3.141592  
#define EPS 1.0e-9
```

100보다는
MAX_SIZE가
이해하기 쉽
지..



#define

- define 명령어는 매크로명을 정의하여 복잡한 상수나 문장을 의미 있는 단어로 사용할 수 있도록 한다.

```
#define 매크로명 확장문자열
```

- 컴파일러는 전처리 과정에서 프로그램에서 사용된 매크로명을 확장문자열로 치환한다.

```
#include <stdio.h>
#define PI 3.14159
```

전처리 후

```
int main()
{
    double radius;

    printf("원의 반지름을 입력하세요 : ");
    scanf("%lf", &radius);

    printf("원의 둘레 : %lf\n", 2*PI*radius);
    printf("원의 면적 : %lf\n", PI*radius*radius);
    return 0;
}
```

// 이곳에는 stdio.h 헤더파일의 내용이 포함된다.

```
int main()
{
    double radius;

    printf("원의 반지름을 입력하세요 : ");
    scanf("%lf", &radius);

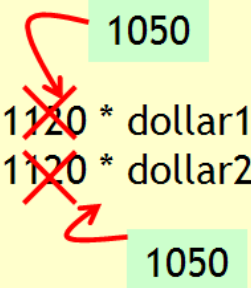
    printf("원의 둘레 : %lf\n", 2*3.14159*radius);
    printf("원의 면적 : %lf\n", 3.14159*radius*radius);
    return 0;
} // 매크로명이 상수값으로 치환
```

단순 매크로의 장점

- 프로그램의 가독성을 높인다.
- 상수의 변경이 용이하다.

```
#include <stdio.h>

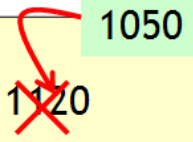
int main(void)
{
    ...
    won1 = 1120 * dollar1;
    won2 = 1120 * dollar2;
    ...
}
```



리터럴 상수를 사용하는 경우:
등장하는 모든 곳을
수정하여야 한다.

```
#include <stdio.h>
#define EXCHANGE_RATE 1120

int main(void)
{
    ...
    won1 = EXCHANGE_RATE * dollar1;
    ...
    won2 = EXCHANGE_RATE * dollar2;
    ...
}
```



기호 상수를 사용하는 경우:
기호 상수가 정의된 곳만 수정
하면 된다.

예제

```
#include <stdio.h>
#define PI 3.14159           // 매크로상수 정의
#define LIMIT 100.0         // 매크로상수 정의
#define MSG "passed!"       // 문자열을 매크로명으로 정의
#define ERR_PRN printf("범위를 벗어났습니다!\n")
                             // 출력문을 매크로명으로 정의

int main()
{
    double radius, area;

    printf("반지름을 입력하세요 : ");
    scanf("%lf", &radius); // 반지름 입력
    area=PI*radius*radius; // 면적 계산
    if(area>LIMIT){         // 면적이 한계값을 넘으면
        ERR_PRN;            // 에러 메시지를 출력한다.
    }
    else{
        printf("원의 면적 : %.2lf (%s)\n", area, MSG);
    }
    // 그렇지 않으면 면적과 문자열 출력

    return 0;
}
```

#define *identifier* *t_string*_{opt}

- 전처리는 문자열을 제외한 파일의 모든 *identifier*를 *token_string*으로 대치

- 예제

```
#define SECONDS_PER_DAY (60 * 60 * 24)
```

- 이 예제에서 *token_string*은 (60 * 60 * 24)이고, 전처리는 그 파일의 이 다음부터 발견되는 기호 상수 SECONDS_PER_DAY 모두를 (60 * 60 * 24)로 대치함

#define *identifier t_string*_{opt}

- #define을 사용하면 프로그램의 명확성과 이식성을 높일 수 있음

```
#define PI 3.14159
```

```
#define C 299792.458 /*speed of light*/
```


#define *identifier t_string*_{opt}

- 특수한 상수들도 기호 상수로 코딩될 수 있음

```
#define    EOF        (-1)
                /* typical end-of-file value */
```

```
#define    MAXINT    2147483647
                /* largest 4-byte integer */
```

`#define identifier t_stringopt`

- 기호 상수는 불명확한 상수를 연상 기호 식별자로 바꿈으로써 프로그램의 문서화에 도움을 줌
- 시스템에 따라 달라지는 상수를 한번에 변경할 수 있으므로 이식성을 높여 줌
- 상수의 실제 값을 검사하는데 한 곳만 검사하면 되므로 신뢰성도 높여 줌

구문 변경

- C의 구문을 사용자의 취향에 맞게 변경하는 것이 가능

- 예제

- ▣ 논리 수식에서 == 대신 EQ 사용
#define EQ ==
- ▣ while 문을 ALGOL 형식의 while do로 변환
#define do /* blank */

인자를 갖는 매크로

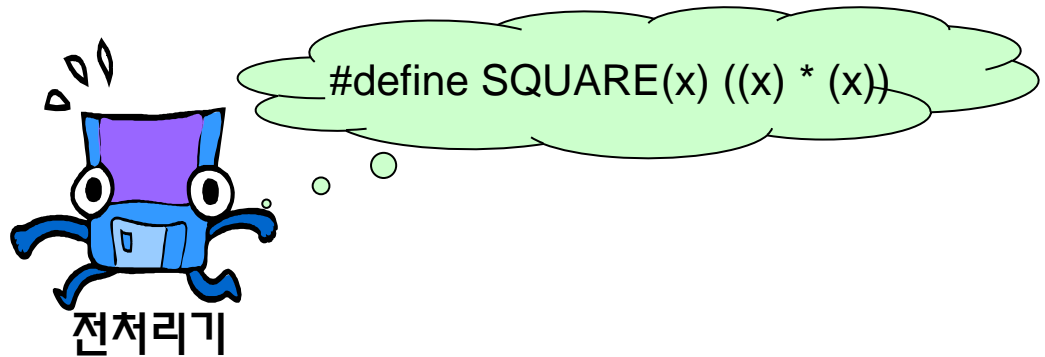
- 일반적인 형태

`#define identifier(id_1, ..., id_n) token_stringopt`

- 첫 번째 *identifier*와 왼쪽 괄호 사이에는 공백이 없어야 함
- 매개변수 목록에는 식별자가 없거나 또는 여러 개가 올 수 있음

함수 매크로 (인자를 갖는 매크로)

- 함수 매크로(*function-like macro*)란 매크로가 함수처럼 매개 변수를 가지는 것
- (예) `#define SQUARE(x) ((x) * (x))`



`v = SQUARE(3);`



`v = ((3)*(3));`

인자를 갖는 매크로

- 동일한 수식이나 문장에 대하여 경우에 따라 각각 다른 값으로 확장될 수 있도록 만든 것이 매크로함수이다.

```
#define 매크로함수명(전달인자) 확장문자열
```

- 전처리 과정에서 전달인자에 따라 서로 다른 문자열로 확장된다.

`#define SUM(x, y) x+y`
 매크로함수명 전달인자 확장문자열

(변수 x, y 사용) `SUM(x, y)` → `x+y`로 치환

(변수 a, b 사용) `SUM(a, b)` → `a+b`로 치환

전달인자가 다르면 치환되는 문자열이 다르다!

```
#define SUM(x, y) x+y
...
int x=10, y=20;
int a=100, b=200;
...
printf("x+y = %d\n", SUM(x, y));
printf("a+b = %d\n", SUM(a, b));
```

함수 매크로의 장단점

- 함수 매크로의 장단점
 - 함수 호출 단계가 필요없어 실행 속도가 빠르다.
 - 소스 코드의 길이가 길어진다.
- 간단한 기능은 매크로를 사용
 - `#define MIN(x, y) ((x) < (y) ? (x) : (y))`
 - `#define ABS(x) ((x) > 0 ? (x) : -(x))`
- 매크로를 한줄 이상 연장하는 방법
 - `#define PRINT(x) if(debug==1 &&`
 - `mode==1)`
 - `printf("%d", x);`

인자를 갖는 매크로

- 예제

```
#define    SQ(x)    ((x) * (x))
```

- $SQ(7 + w)$ 는 $((7 + w) * (7 + w))$ 로 확장
- $SQ(SQ(*p))$ 는 $((((*p) * (*p))) * (((*p) * (*p))))$ 로 확장

인자를 갖는 매크로 – 유의 사항

- 매크로를 정의할 때 올바른 평가 순서를 유지하기 위해 괄호를 적절히 사용해야 함
- 부적절한 괄호를 사용한 예

```
#define    SQ(x)    x * x
```

```
: SQ(a + b)
```

$\Rightarrow a + b * a + b \neq ((a + b) * (a + b))$

```
#define    SQ(x)    (x) * (x)
```

```
: 4 / SQ(2)
```

$\Rightarrow 4 / (2) * (2) \neq (4 / ((2) * (2)))$

인자를 갖는 매크로 – 유의 사항

- 실수할 수 있는 또 다른 예제

```
#define  SQ  (x)  ((x) * (x))
:  SQ(7)  ==> (x)  ((x) * (x)) (7)
```

```
#define  SQ(x)  ((x) * (x));    /* error */
:  if  (x == 2) x = SQ(y);
    else  ++x;
==>
    if  (x == 2) x = ((y) * (y));;
    else  ++x;
```

인자를 갖는 매크로 – 유의 사항

- 매크로함수는 확장 후에 다른 연산자와의 추가적인 연산에 의해서 부작용이 생길 수 있다.

```
#define MUL(x, y) x*y
```

```
printf(“%d\n”, 30/MUL(2, 5)); // 전처리 되기 전의 코드
```

```
printf(“%d\n”, 30/2*5); // 전처리 된 이후의 코드
```

↘ 나눗셈이 먼저 연산 된다!

- 확장 후의 부작용을 막기 위해서 확장문자열에 괄호를 사용한다.

```
#define MUL(x, y) ((x)*(y)) // 확장문자열의 각 문자를 모두 괄호로 감싼다.
```

```
printf(“%d\n”, 30/MUL(2, 2+3));
```

↘ `printf(“%d\n”, 30/((2)*(2+3)));`

예제

```
#include <stdio.h>
#define PI 3.14159           // 매크로상수 정의
#define SQUARE(x) ((x)*(x)) // 제곱을 구하는 매크로함수 정의

int main()
{
    double radius;

    printf("반지름을 입력하세요 : ");
    scanf("%lf", &radius);
    printf("원의 면적은 : %.2lf", PI*SQUARE(radius));
    return 0;    // 매크로상수와 매크로함수를 사용하여 원의 면적을 구한다.
}
```

인자를 갖는 매크로

- 매크로를 정의할 때, 매크로 몸체에 매크로나 함수를 사용할 수 있음

```
#define min(x, y)      (((x) < (y)) ? (x) : (y))
```

```
#define min4(a, b, c, d)  min(min(a, b), min(c, d))
```

```
#define SQ(x)           ((x) * (x))
```

```
#define CUBE(x)          (SQ(x) * (x))
```

```
#define F_POW(x)        sqrt(sqrt((CUBE(x))))
```

인자를 갖는 매크로

- **#undef는 매크로 정의를 무효화함**

`#undef identifier`

- **전처리기 결과 보기**

`cc -E file.c`

- ▣ 이 명령을 사용하면, 전처리기가 작업을 수행한 다음에 더 이상의 컴파일이 일어나지 않음

stddef.h

- 이 헤더 파일은 다른 곳에서 공통적으로 사용되는 몇 가지 형 정의와 매크로를 포함하고 있음
- 예

```
typedef int    ptrdiff_t; /* pointer difference type */
typedef short  wchar_t;   /* wide character type */
typedef unsigned size_t;  /* the sizeof type */
#define NULL    ((void *) 0)
```

stdio.h의 매크로

- **getc()와 putc() 함수**

- getc() : 파일로부터 한 문자를 읽음
- putc() : 파일에 한 문자를 출력

- **getchar()와 putchar() 매크로**

```
#define    getchar()    getc(stdin)
```

```
#define    putchar(c)   putc((c), stdout)
```


ctype.h의 매크로

매크로	참 값(0이 아닌 값)이 리턴되는 경우
isalpha(c)	c가 문자일 때
isupper(c)	c가 대문자일 때
islower(c)	c가 소문자일 때
isdigit(c)	c가 숫자일 때
isalnum(c)	c가 영문자나 숫자일 때
isxdigit(c)	c가 16진 숫자일 때
isspace(c)	c가 공백 문자일 때
ispunct(c)	c가 구두 문자일 때
isprint(c)	c가 인쇄가능한 문자일 때
isgraph(c)	c가 공백이 아닌 인쇄가능한 문자일 때
isctrl(c)	c가 제어 문자일 때
isascii(c)	c가 ASCII 코드일 때

ctype.h의 매크로

함수 또는 매크로	리턴되는 값
<code>toupper(c)</code> <code>tolower(c)</code> <code>toascii(c)</code>	<code>c</code> 가 소문자이면 대응되는 대문자, 아니면 <code>c</code> <code>c</code> 가 대문자이면 대응되는 소문자, 아니면 <code>c</code> <code>c</code> 와 대응되는 ASCII 코드 값

조건부 컴파일

- 조건부 컴파일을 위한 지시자

```
#if      constant_integral_expression
```

```
. . .
```

```
#endif
```

```
#ifdef   identifier      // or #if  
        defined(identifier)
```

```
. . .
```

```
#endif
```

```
#ifndef  identifier
```

```
. . .
```

```
#endif
```

조건부 컴파일

- 코드가 컴파일되기 위해서는 **#if** 다음의 상수 수식이 영이 아닌 값(참)을 가져야 함
- **#ifdef**나 **#if defined**와 **#endif** 사이의 코드가 컴파일되기 위해서는 **identifier**가 이미 정의되어 있어야 하고, 그 **identifier**가 취소(**#undef identifier**)되지 않았어야 함
- **#ifndef**와 **#endif** 사이의 코드가 컴파일되기 위해서는 **#ifndef** 다음의 **identifier**가 현재 정의되어 있지 않아야 함

조건부 컴파일 예제

- 이식성이 높은 코드를 만들기 위해 사용할 수 있음

```
#if defined(HP9000) || defined(SUN4)
    && !defined(VAX)
        .....    /* machine-dependent code */
#endif
```

조건부 컴파일 예제

- 디버깅을 위해 사용할 수 있음

```
#define DEBUG 1  
....  
#if DEBUG  
    debugging code  
#endif
```

```
#define DEBUG  
....  
#ifdef DEBUG  
    debugging code  
#endif
```

조건부 컴파일 예제

- 디버깅을 위해 사용할 수 있음

statements

*/**

more statements

**/*

and still more statements

statements

#if 0

more statements

#endif

and still more statements

조건부 컴파일 예제

- 매크로 이름이 중복 지정되는 것을 피하기 위해 사용할 수 있음

```
#include "everything.h"
```

```
#undef    PIE
```

```
#define   PIE    "I like apple."
```

```
.....
```


조건부 컴파일

- **if-else** 형의 구조

```
#if
```

```
#elif constant_integral_expression |
```

```
#else
```

```
#endif
```

미리 정의된 매크로

- ANSI C에는 미리 정의된 다섯 개의 매크로가 있고, 이 매크로는 항상 사용할 수 있으며, 프로그래머가 정의를 해제할 수 없음
- 각 매크로 이름은 두개의 밑줄문자로 시작해서 두개의 밑줄문자로 끝남

미리 정의된 매크로	값
__DATE__	현재 날짜를 포함하는 문자열
__FILE__	파일 이름을 포함하는 문자열
__LINE__	현재 라인 번호를 나타내는 정수
__STDC__	ANSI C 표준을 따르는 경우 0이 아닌 값을 가짐
__TIME__	현재 시간을 포함하는 문자열

연산자

- "문자열화" 연산자

```
#define message_for(a, b) \
    printf("#a " and " #b ": We love you!\n")
```

```
int main(void)
{
    message_for(Carole, Debra);
    return 0;
}
```

연산자

- 토큰 결합 연산자

```
#define    x(i)    x ## i  
x(1) = x(2) = x(3);
```

- ▣ 전처리기 수행 후의 결과:

```
x1 = x2 = x3;
```

assert() 매크로

- 수식의 값이 기대하고 있는 값인가를 확인할 때 사용

```
#include <assert.h>
void f(char *p, int n){
    .....
    assert(p != NULL);
    assert(n > 0 && n < 7):
    .....
```

- 만일 단정이 실패하면, 시스템은 메시지를 출력하고 프로그램을 중단함
- NDEBUG가 정의되어 있으면, 모든 단정은 무시됨

#error

- 조건들을 강요하기 위해 사용
- 전처리가 #error를 만나면, 컴파일 오류가 발생하고, 이 지시자 다음에 쓰인 문자열이 화면에 출력됨
- 사용 예

```
#if A_SIZE < B_SIZE  
    #error "Incompatible sizes"  
#endif
```

행 번호

- **사용 형태**

#line integral_constant "filename"

- ▣ 이것은 컴파일러로 하여금 원시 프로그램의 행 번호를 다시 매기게 하여 그 다음 행의 번호가 integral_constant가 되게 함
- ▣ 또한 컴파일러에게 현재 원시 파일의 이름이 filename이라고 믿게 함
- ▣ 파일 이름이 명시되지 않으면, 행 번호만을 다시 매김
- ▣ 보통 행 번호는 프로그래머에게 감추어지고, 경고나 구문 오류가 발생할 때에만 나타남

대응 함수

- 표준 헤더 파일에 있는 매개변수를 갖는 대부분의 매크로는 이와 대응되는 함수를 표준 라이브러리에 가지고 있음
- 매크로 대신 함수를 사용하기 위해서는 다음과 같이 함
 - 방법 1 - 함수 사용 전에 다음과 같은 행을 삽입함
`#undef isalpha`
 - 방법 2 - 다음과 같이 호출함
`(isalpha)(c)`

질의 및 응답