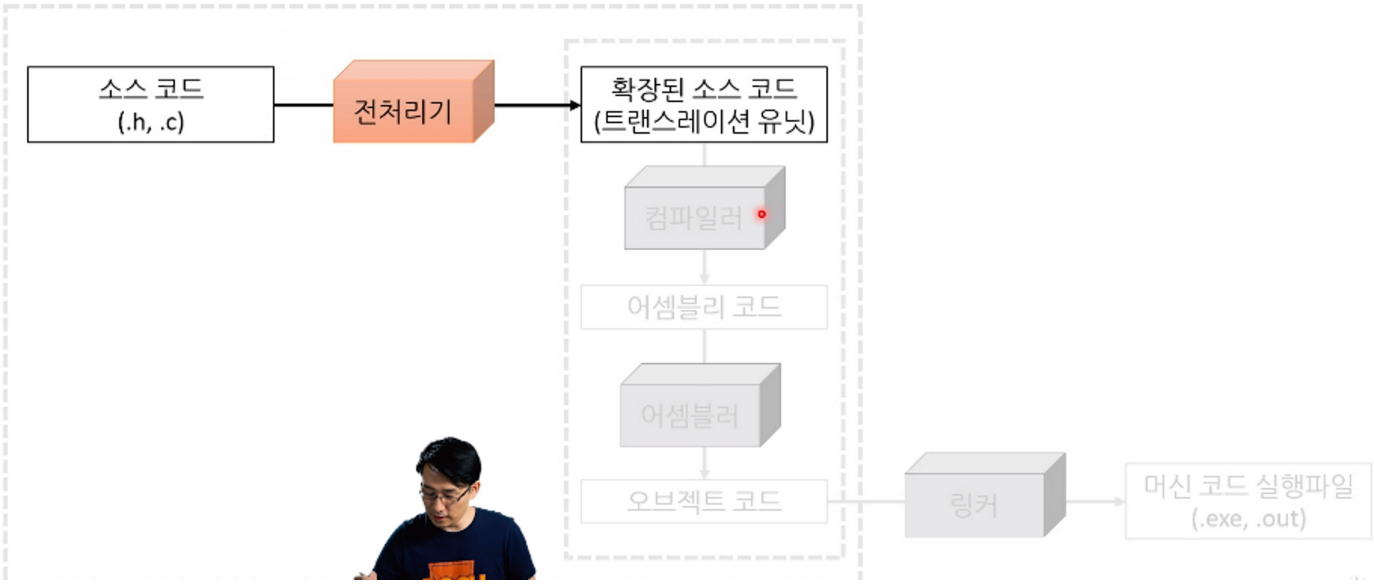
# C 전처리기

2021년 3월 18일 목요일 오후 8:28

- 전처리 단계 복습




- 전처리 단계란?
  - C 파일에서 주석을 날려버리고
  - 매크로를 확장하고
  - 인클루드에 있는 h파일을 복붙해서
  - 트랜슬레이션 유닛으로 바꾸는 과정이었다
- 전처리기로 할 수 있는 일들
  1. 다른 파일을 인클루드
     - 전처리기 지시문 #include 사용
  2. 매크로를 다른 텍스트로 대체
     - 전처리기 지시문 #define, #undef와 전처리기 연산자 #, ## 사용
  3. 소스 파일의 일부를 조건부로 컴파일
     - 전처리기 지시문 #if, #ifdef, #ifndef, #else, #elif, #endif 사용
  4. 일부러 오류를 발생시킴
     - 전처리기 지시문 #error 사용
- 매크로 대체와: #define

```
#define 식별자 대체_목록(선택)
```

  - #define A (10)
    - 전처리기가 소스 코드 뒤지다가 A가 보이면 모두 (10)으로 바꿔줌
  - #define A
    - 이것도 가능
    - 하지만 바꿔줄 내용이 없음
    - 그 대신 다른 전처리기 지시어로 A가 정의(define) 돼 있는지 판단 가능
    - 조건부 컴파일에서 좀 더 자세히 볼 내용
  - 

```
#define TRUE (1)
#define FALSE (0)
```

    - 많이 본 것
  - 

```
#define 식별자(매개변수) 대체_목록
```

    - 심지어 함수처럼 쓰는 것도 가능
      - 이 경우는 매크로 함수라고 함
      - 이건 뒤에서 설명할 예정
- 매크로 대체: #undef

### #undef 식별자

- 이미 정의된 식별자를 없애는 것
- 해당 식별자로 정의된 텍스트 매크로가 없다면 이 지시문은 무시됨

```
#define NUMBER (20)
#undef NUMBER
```

```
/* 메인 함수 */
printf("%d\n", NUMBER); /* 컴파일 오류 */
```

실제로 이렇게 무식한 코드는 없습니다
말 그대로 "보여주기 위한 코드"

## 매크로 대체: 미리 정의되어 있는 #define

- 모든 C 구현이 정의하는 것들
  - __FILE__: 현재 파일의 이름을 문자열로 표시
  - __LINE__: 현재 코드의 줄 번호를 정수형으로 표시
  - 두 매크로 모두 오류 출력 시 자주 사용

```
fprintf(stderr, "internal error: %s, line %d.\n", __FILE__, __LINE__);
```

```
internal error: main.c, line 27.
```

  - (C95부터 지원) __STDC_VERSION__: 현재 컴파일에 사용중인 C 표준
- 당연히 각 컴파일러가 자기 맘대로 정의하는 것들도 있음

## 조건부 컴파일

- 조건이다보니 if/else문과 유사한 지시문들이 대거 포진
- 조건에 따라 특정 부분의 코드를 컴파일에 포함 또는 배제
- #if 표현식
- #ifndef 식별자 혹은 #if defined 식별자
- #ifndef 식별자 혹은 #if !defined 식별자
- #elif 표현식
- #else
- #endif

## 조건부 컴파일: 인클루드 가드

- 예전에 배운 인클루드 가드 기억?
- 순환 헤더 인클루드, 즉, 헤더 꼬임을 방지하는 방법
  1. 어떤 상수를 #define으로 정의함
  2. 그 후 컴파일러에게 조건적으로 코드를 컴파일하라고 지시

```
#ifndef FOO_H
#define FOO_H
/* 원래 헤더 파일 내용 */
#endif /* FOO_H */
```

만약 FOO_H가 정의되지 않았다면
FOO_H를 정의할 것
그리고 이 코드를 원래대로 포함시킬 것
#ifndef 블록의 끝

  - 조건적으로 한번만 인클루드 하게 됨

## 어떤 식별자가 #define 되어있는지 판단

```
#ifndef NULL
#define NULL (0)
#endif
```

NULL이 정의 안 됐다면,
값이 0인 NULL을 정의

```
#if !defined(NULL)
#define NULL (0)
#endif
```

NULL이 정의 안 됐다면,
값이 0인 NULL을 정의

```
#if defined(NULL)
#undef NULL
#endif
```

NULL이 정의됐다면,
그 정의를 해제시킴

```
#define NULL (0)
```

값이 0인 NULL을 정의

- NULL 중복정의 방지

- 조건부 컴파일에서 주의할 점

```
#define A

#if defined(A)      /* 참 */
#define LENGTH (10)
#endif

#if A               /* 거짓 */
#define LENGTH (10)
#endif
```

- 언뜻 보면 둘다 참일 것 같지만 그렇지 않음에 주의
  - A가 0이면 거짓 0이 아니면 참
- 조건부 컴파일: 버전 관리
  - 새 기능을 추가 중일 때, 버전 관리용으로 사용할 수 있음
    - 새 버전을 만들고 싶은데 기존 버전은 없애고 싶지 않을 때

```
int spawn_monster(...)
{
    get_monster_skin();
    get_monster_stat();

#if defined(FILE_VERSION_2)
    use_custom_skin(...);
#endif
    calculate_spawn_location();
    return TRUE;
}
```

    - 파일 버전 1로 돌아가고 싶으면 정의한 FILE_VERSION_2를 없앰
  - #elif와 #else를 사용해서 각 버전마다 필요한 작업을 할 수 있음

```
...

#if defined(FILE_VERSION_2)
    use_custom_skin(...);
#elif defined(FILE_VERSION_3)
    use_custom_voice(...);
#else
    use_default_skin(...);
    use_default_voice(...);
#endif

...
```

    - 파일 버전 2, 3이면(#if, #elif) 실행 그것도 아니면(#else) 나머지 실행
- 조건부 컴파일: 주석 처리를 편하게
  - 지금까지 사용하던 주석 처리 방법
    - /* */
  - #if 0와 #endif를 사용하면 보다 편하게 주석 처리가 가능

```
#if 0
int has_numbers(int n)
{
    int i = 0;
    for (i = 0; i < LENGTH; ++i) {
        if (s_numbers[i] == n) {
            return TRUE;
        }
    }
    return FALSE;
}
#endif
```

    - #if 0은 항상 거짓이므로 밑에 코드 실행 안됨

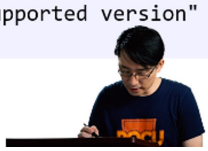
- 컴파일 오류 발생

  `#error 메세지`

  - 컴파일 도중에 강제로 오류를 발생시키는 매크로
  - 메시지를 꼭 따옴표로 감쌀 필요는 없음

```c
/* version.h */
#define VERSION 10
```

```c
/* builder.h */
#if VERSION != 11
#error "unsupported version"
#endif
```



```
builder.h:8:2: error: "unsupported version"
#error "unsupported version"
 ^
```

- 컴파일 중에 매크로 정의하기
  - 컴파일 도중에 -D 옵션으로 전달 가능

```
> clang -std=c89 -W  -Wall  -pedantic-errors -DA *.c
```

    - #define A (1)과 똑같은 결과 (#define A가 아님)
  - 직접 대체할 값 지정할 수 있음

```
> clang -std=c89 -W  -Wall  -pedantic-errors -DA=52 *.c
```

    - #define A (52)와 똑같은 결과
- 배포용으로 컴파일하기: -DNDEBUG

```
> clang -std=c89 -W  -Wall  -pedantic-errors -DNDEBUG *.c
```

  - 배포(release) 모드로 실행파일을 컴파일하라고 알려주는 매크로
    - NDEBUG: '디버그가 아니다(not debug)'란 뜻
    - Assert()가 사라짐
    - 디버그 모드에서만 실행될 코드는 #if !defined(NDEBUG) 속에 넣을 것
  - 이 대신 다음과 같은 매크로를 직접 정의해 사용하는 프로젝트 많음
    - DEBUG: 디버그용 빌드
    - RELEASE: 배포용 빌드
    - 기타: 필요에 따라 다양한 빌드를 지정
- 매크로 함수
  - #define을 할 때 '대체 가능한 매개변수 목록'을 받음

```c
#define SQUARE(a) a * a
#define ADD(a,b) a + b
```

```c
/* 메인 함수 */
int num1;
int num2;
int result;

num1 = 10;
num2 = 20;
result = ADD(num1, num2); /* 30 */
result = SQUARE(num1);      /* 100 */
```

- 매크로 함수에서 흔히 하는 실수
  - 결과가 무엇일까요?

```c
#define SQUARE(a) a * a
#define ADD(a,b) a + b
```

```c
/* 메인 함수 */
int num1;
int num2;
int result;

num1 = 10;
num2 = 20;
result = 10 * ADD(num1, num2); /* ?? */
```

        - num1 + num2로 그대로 복사가 됨 (괄호가 쳐지지 않고)
            - 10 * num1 + num2
                - 결과는 300
- 베스트 프랙티스: 매크로에 소괄호를 쓰자
    - #define ADD(a, b) a + b
    - #define ADD(a, b) (a+ b)
- 매크로가 여러 줄이면 어떡하죠?
    - \ 를 사용하면 매크로를 여러 줄로 나눌 수 있음

```
#define POW(n,p,i,r) r = 1;                     \
                     for (i = 0; i < p; ++i) { \
                         r *= n;                \
                     }                          \
```

```
int num;
int exp;
int result;
int i;

num = 2;
exp = 10;
POW(num, exp, i, result); /* result: 1024 */
```

- 매크로 함수의 활용: 어서트 재정의
    - 어셈블리 코드를 이용한 나만의 어서트 매크로를 만들 수 있음

```
#define ASSERT(condition, msg)                                      \
    if (!(condition)) {                                             \
        fprintf(stderr, "%s(%s: %d)\n", msg, __FILE__, __LINE__); \
        __asm { int 3 }                                             \
    }                                                               \
```

```
/* 메인 함수 */
int month = 20;
ASSERT(month < 12, "invalid month number");
```

    - 왜 어서트 매크로를 대신 사용할까?
        - assert( )는 실패 시 호출 스택(call stack)의 현재 위치가 assert( ) 함수 속
            - 디버깅 시 원래 코드를 보기 불편함
        - __asm { int 3 }는 실제로 어서트에 실패한 코드가 호출 스택의 현 위치
        - 또한 사람이 읽기 편한 설명도 눈에 딱 보임 (stderr 출력은 필수 아님)
            - ASSERT(month < 12, "invalid month number");
            - assert(month < 12);
        - 단, int 3는 x86 어셈블리에서 프로그램 실행을 중지하는 인터럽트
            - -- debug break; 도 같은 역할
        - 플랫폼마다 사용하는 어셈블리 명령어가 달라짐
- 전처리기 명령어: #명령어

```
#define 식별자(매개변수) 대체_목록
```

    - 매개변수 자체를 문자열로 바꿔줌
    - 매개변수를 쌍따옴표로 감싸는 것

```
#define str(s) #s
```

```
printf("%s\n", str(\n));
printf("%s\n", str("\n"));
printf("%s\n", str(int main));
printf("%s\n", str("Hello World"));
printf("%s\n", str(num1));
```

```
"\n"
int main
"Hello World"
num1
```

    - 가끔만 쓸일 이있다

- 전처리기 명령어: ## 명령어

  ```
  #define 식별자(매개변수) 대체_목록
  ```

  - 이건 쓸 일이 많을 것임
  - 대체 목록 안에 있는 두 단어를 합쳐서 새로운 텍스트로 바꿈
    - 단어는 매개변수일 수도 아닐 수도 있음
  - #과 달리 문자열 데이터를 만들어주는게 아님

  ```
  #define print(n) printf( "%d\n", g_id_##n )
  ```

  ```
  int g_id_none = 0;
  int g_id_teacher = 1;
  int g_id_student = 2;
  ```

  ```
  /* 메인 함수 */
  print(number);  /* 컴파일 오류: g_id_number라는 변수가 없음 */
  print(none);    /* 컴파일: g_id_none의 값인 0 출력 */
  print(student); /* 컴파일: g_id_student의 값인 2 출력 */
  ```

    - #define print를 소문자로 썼는데
      매크로의 이름은 대문자로 쓰는게 일반적임

- # vs ##

```
#define combine1(a, b) (a#b)
#define combine2(a, b) (a##b)
```

```
/* 메인 함수 */
int student_id = 987654;

// 컴파일 오류: student_"id"
printf("%d\n", combine1(student_, id));

// 컴파일: student_id의 값인 987654를 출력
printf("%d\n", combine2(student_, id));
```

- 매크로 함수의 장점과 단점
  - 장점
    - 함수 호출이 아닌 곧바로 코드를 복붙하는 개념
    - 함수 호출에 따른 과부하가 없음
    - c에서 불편한 것들 중 일부는 매크로 꼼수로 해결 가능
  - 단점
    - 디버깅이 아주 매우 어려움
    - \를 사용해서 아무리 읽기 좋게 매크로 함수를 만들어도 중단점 사용 불가