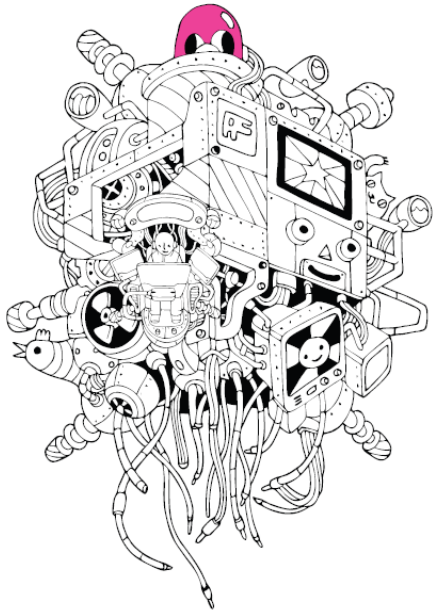
윤성우의
열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

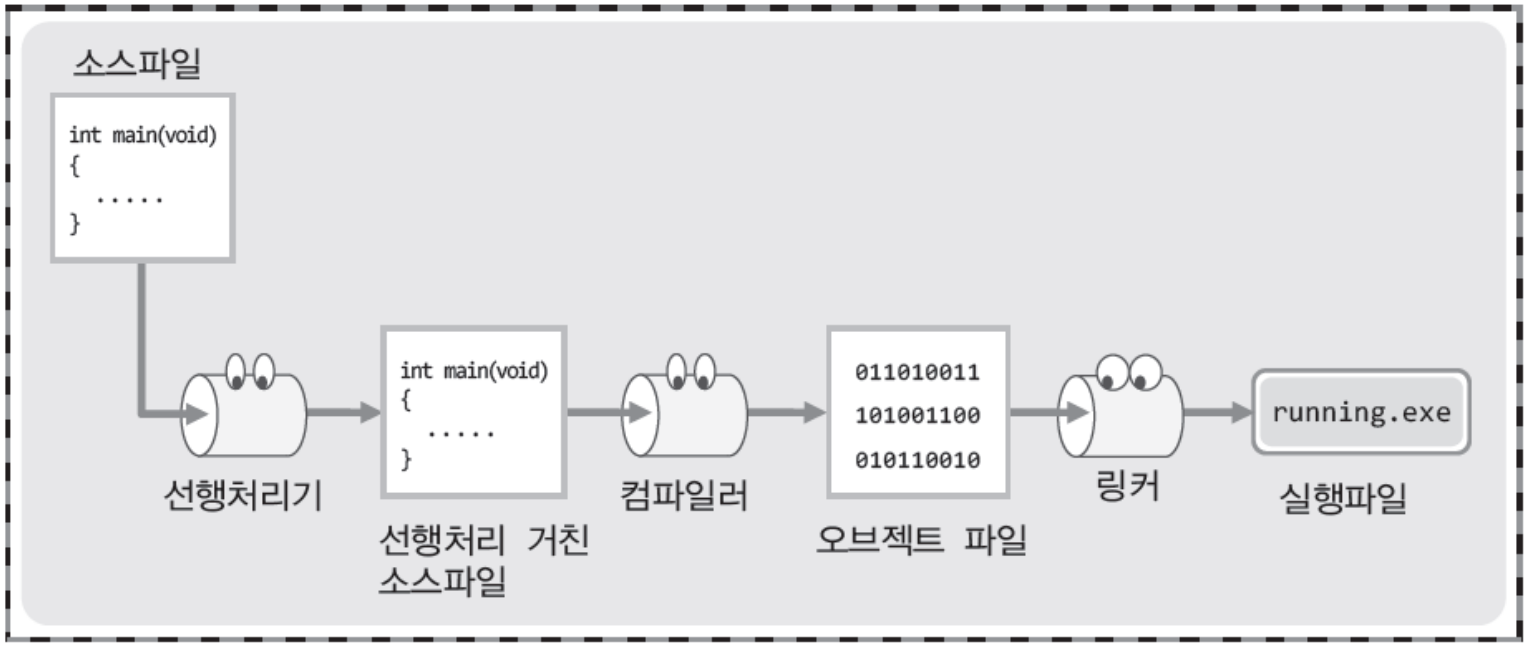
Chapter 26. 매크로와 선행처리기

윤성우의
열혈 C 프로그래밍

Chapter 26-1. 선행처리기와 매크로

윤성우 저 열혈강의 C 프로그래밍 개정판

선행처리는 컴파일 이전의 처리를 의미합니다.

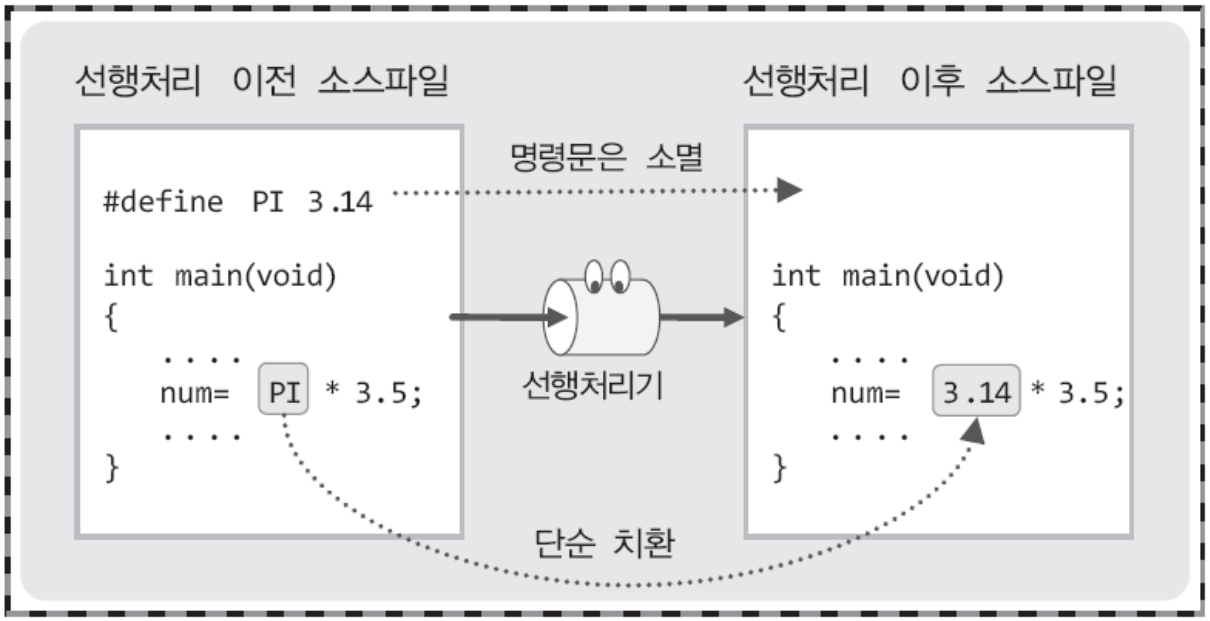


일반적으로 컴파일의 과정에 포함이 되어 이야기하지만, 선행처리의 과정과 컴파일의 과정은 구분이 된다.

선행처리기의 일 간단히 맛보기

#define PI 3.14 가 의미하는 바는 다음과 같다.

"PI를 3.14로 치환하라!" .



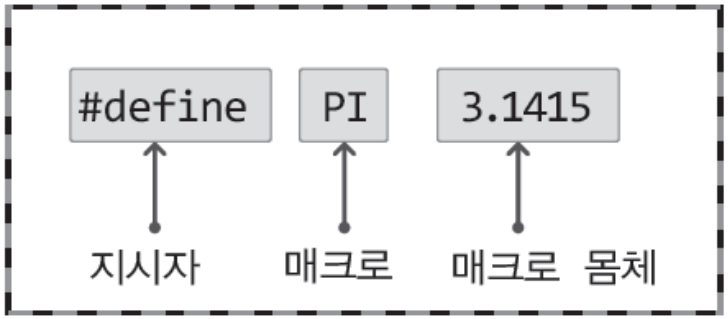
컴파일러에 비해서 선행처리기의 역할은 매우 간단하다. 쉽게 말해서 '단순한 치환'의 작업을 거친다. 그리고 선행처리기에게 무엇인가를 명령하는 문장은 #으로 시작한다.

윤성우의
열혈 C 프로그래밍

Chapter 26-2. 대표적인 선행처리 명령문

윤성우 저 열혈강의 C 프로그래밍 개정판

#define: Object-like macro



PI를 3.1415로 무조건 치환하라!

```
#define NAME        "홍길동"
#define AGE         24
#define PRINT_ADDR    puts("주소: 경기도 용인시\n");
int main(void)
{
    printf("이름: %s \n", NAME);
    printf("나이: %d \n", AGE);
    PRINT_ADDR;
    return 0;
}
```

```
printf("이름: %s \n", "홍길동");
printf("나이: %d \n", 24);
puts("주소: 경기도 용인시 \n");
```

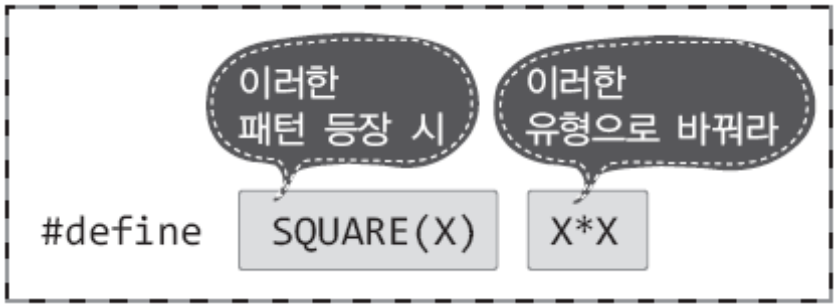
치환된 결과물!

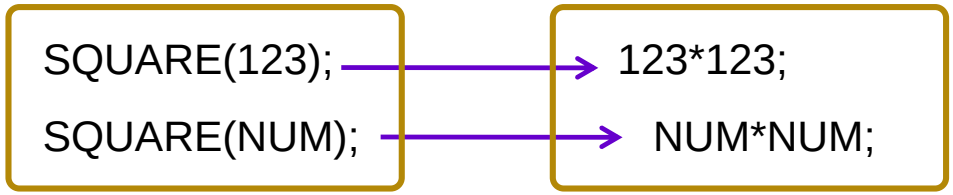
실제 컴파일 되는 문장들!

실행결과

```
이름: 홍길동
나이: 24
주소: 경기도 용인시
```

#define: Function-like macro





SQUARE(X) 패턴에 부합!

따라서 치환!

이러한 변환의 과정을 가리켜
'매크로 확장(macro expansion)'이라 한다.

매크로 확장의 예

```
#define SQUARE(X) X*X
int main(void)
{
    int num=20;

    /* 정상적 결과 출력 */
    printf("Square of num: %d \n", SQUARE(num));
    printf("Square of -5: %d \n", SQUARE(-5));
    printf("Square of 2.5: %g \n", SQUARE(2.5));

    /* 비정상적 결과 출력 */
    printf("Square of 3+2: %d \n", SQUARE(3+2));
    return 0;
}
```

3과 2의 합인 5가 SQUARE 함수의 인자가 되어, 25가 출력되는 것이 상식적이다! 그러나 출력결과는 다르다!

```
Square of num: 400
Square of -5: 25
Square of 2.5: 6.25
Square of 3+2: 11
```

실행결과

잘못된 매크로의 정의와 소괄호의 해결책

#define SQUARE(X) X*X

SQUARE(3+2) → 3+2*3+2 → 11 오류의 원인

SQUARE((3+2)) → (3+2)*(3+2) → 25 사용하기 불편

#define SQUARE(X) (X)*(X)

SQUARE(3+2) → (3+2)*(3+2) → 25 이 경우는 해결

num=120/SQUARE(2) → 120 / (2)*(2) 여전히 문제!

#define SQUARE(X) ((X)*(X))

num=120/SQUARE(2) → 120 / ((2)*(2)) 최상의 해결책!

매크로를 두 줄에 걸쳐서 정의하는 방법

```
#define SQUARE(X)
((X)*(X))
```

이렇듯 두 줄에 걸쳐서 매크로를 정의하면 에러가 발생한다!

```
#define SQUARE(X)   \
((X)*(X))
```

첫 번째 줄의 끝에 \을 삽입하면 에러가 발생하지 않는다. 이는 \이 매크로의 정의가 이어짐을 뜻하기 때문이다.

먼저 정의된 매크로의 사용

```
#define PI 3.14
#define PROUDCT(X, Y)  ((X)*(Y))
#define CIRCLE_AREA(R) (PROUDCT((R), (R))*PI)

int main(void)
{
    double rad=2.1;
    printf("반지름 %g인 원의 넓이: %g \n", rad, CIRCLE_AREA(rad));
    return 0;
}
```

```
반지름 2.1인 원의 넓이: 13.8474
```

실행결과

위 예제에서 보이듯이, 앞 줄에서 먼저 정의된 매크로는 새로운 매크로를 정의하는데 있어서 사용될 수 있다.

일반함수와 비교한 매크로 함수의 장점

장점 1.

매크로 함수는 일반 함수에 비해 실행속도가 빠르다.

함수의 호출을 완성하기 위해서는 별도의 메모리 공간이 필요하고 호출된 함수로의 이동 및 반환의 과정을 거쳐야 한다. 반면 매크로 함수는 정의된 몸체로 치환이 이뤄지니 이러한 일이 불필요하며, 때문에 실행속도가 빨라질 수 밖에 없다.

장점 2.

자료형에 따라서 별도로 함수를 정의하지 않아도 된다.

전달되는 인자의 자료형에 구분을 받지 않으므로 자료형에 의존적이지 않다.

매크로 함수의 단점

단점

1. 정의하기가 정말로 까다롭다.
2. 디버깅하기가 쉽지 않다. 선행처리 후 컴파일러에 의해서 에러가 감지되므로

매크로 함수로 정의하면?

```
int DiffABS(int a, int b)
{
    if(a>b)
        return a-b;
    else
        return b-a;
}
```

잘못 정의된 매크로

```
#define DIFF_ABS(X, Y)  ( (x)>(y) ? (x)-(y) : (y)-(x) )
int main(void)
{
    printf("두 값의 차: %d \n", DIFF_ABS(5, 7));
    printf("두 값의 차: %g \n", DIFF_ABS(1.8, -1.4));
    return 0;
}
```

그러나 컴파일러는 컴파일 된 이후의 내용을 기준으로 오류를 이야기한다.

함수를 매크로로 정의하기 위한 조건

조건 1.

작은 크기의 함수

한 두줄 정도 크기의 작은 함수가 아니면 매크로로 정의하는 것이 쉽지 않아서 오류발생 확률이 높아진다. 그리고 if~else, for와 같이 실행의 흐름을 컨트롤 하는 문장도 매크로로 정의하기 쉽지 않다.

조건 2.

호출의 빈도수가 높은 함수

함수를 매크로로 정의하는 데에는 성능적 측면이 고려된다. 그런데 호출의 빈도수가 높지 않으면 애써 매크로로 함수를 정의하는 수고를 할 이유가 줄어든다.

윤성우의

열혈 C 프로그래밍

Chapter 26-3. 조건부 컴파일을 위한 매크로

윤성우 저 열혈강의 C 프로그래밍 개정판

#if . . . #endif : 참이라면

```
#define ADD   1
#define MIN   0

int main(void)
{
    int num1, num2;
    printf("두 개의 정수 입력: ");
    scanf("%d %d", &num1, &num2);
#if ADD    // ADD가 '참'이라면
    printf("%d + %d = %d \n", num1, num2, num1+num2);
#endif
#if MIN    // MIN이 '참'이라면
    printf("%d - %d = %d \n", num1, num2, num1-num2);
#endif
    return 0;
}
```

선행처리 과정에서 ADD가 '참'이면 ~endif 까지 컴파일 대상에 포함

선행처리 과정에서 MIN이 '참'이면 ~endif 까지 컴파일 대상에 포함

실행결과

```
두 개의 정수 입력: 5 4
5 + 4 = 9
```

#ifdef . . . #endif : 정의되었다면

```
// #define ADD   1
#define MIN   0

int main(void)
{
    int num1, num2;
    printf("두 개의 정수 입력: ");
    scanf("%d %d", &num1, &num2);

#ifdef ADD    // 매크로 ADD가 정의되었다면
    printf("%d + %d = %d \n", num1, num2, num1+num2);
#endif

#ifdef MIN    // 매크로 MIN이 정의되었다면
    printf("%d - %d = %d \n", num1, num2, num1-num2);
#endif

    return 0;
}
```

ADD와 MIN의 정의 여부만 중요한 상황이라면 각각의 매크로에 값을 지정할 필요가 없다. 즉, 다음과 같이 정의해도 된다.

```
#define ADD
#define MIN
```

ADD가 정의되었다면 ~endif 까지 컴파일 대상에 포함

MIN이 정의되었다면 ~endif 까지 컴파일 대상에 포함

실행결과

```
두 개의 정수 입력: 7 2
7 - 2 = 5
```

#ifndef . . . #endif : 정의되지 않았다면

```
#ifndef ADD
    printf("%d + %d = %d ₩n", num1, num2, num1+num2);
#endif
```

ADD가 정의되지 않았다면 ~endif 까지 컴파일 대상에 포함

```
#ifndef MIN
    printf("%d - %d = %d ₩n", num1, num2, num1-num2);
#endif
```

MIN이 정의되지 않았다면 ~endif 까지 컴파일 대상에 포함

#else의 삽입: #if, #ifdef, #ifndef에 해당

```
#define HIT_NUM   5

int main(void)
{
#if HIT_NUM==5
    puts("매크로 상수 HIT_NUM은 현재 5입니다.");
#else
    puts("매크로 상수 HIT_NUM은 현재 5가 아닙니다.");
#endif
    return 0;
}
```

포함 조건

매크로 HIT_NUM이 5가 아닐 경우 포함되는 문장

실행결과

```
매크로 상수 HIT_NUM은 현재 5입니다.
```

#else를 추가해서

조건이 '참'이 아닌 경우에 컴파일의 대상에 포함시킬 문장들을 구성할 수 있다.

#elif의 삽입: #if에만 해당

```
#define HIT_NUM  7

int main(void)
{
#if HIT_NUM==5
    puts("매크로 상수 HIT_NUM은 현재 5입니다.");
#elif HIT_NUM==6
    puts("매크로 상수 HIT_NUM은 현재 6입니다.");
#elif HIT_NUM==7
    puts("매크로 상수 HIT_NUM은 현재 7입니다.");
#else
    puts("매크로 상수 HIT_NUM은 5, 6, 7은 확실히 아닙니다.");
#endif

    return 0;
}
```

조건에 따라서 이중에서 하나의 문장만 컴파일의 대상으로 포함이 된다.

```
매크로 상수 HIT_NUM은 현재 7입니다.
```

실행결과

#elif와 #else를 추가해서 if ~ else if ~ else 구문과 동일한 형태를 구성할 수 있다.

#ifdef 활용

```
int main()
{
    int n = 3;

#ifdef NEVER_DEFINE
    n = 5;
    printf("never define!₩n");
#endif

    printf("n: %d₩n", n);
}
```

주석으로 활용 가능

```
n: 3
```

실행결과

정의하지 않은 이름의 매크로를 사용하여 주석으로 처리할 수 있다.

윤성우의
열혈 C 프로그래밍

Chapter 26-4. 매개변수의 결합과 문자열화

윤성우 저 열혈강의 C 프로그래밍 개정판

문자열 내에서는 매크로의 매개변수 치환 불가

[문자열 치환을 목적으로 정의된 매크로]

```
#define    STRING_JOB(A, B)        "A의 직업은 B입니다."
```

치환의 대상이 되는 A와 B가 문자열 안에 존재함에 주목하자!

[매크로의 확장 결과]

```
STRING_JOB(이동춘, 나무꾼)        "A의 직업은 B입니다."
STRING_JOB(한상순, 사냥꾼)        "A의 직업은 B입니다."
```

이동춘과 나무꾼, 그리고 한상순과 사냥꾼이 기대대로 치환되지 않았다. 문자열 안에서는 치환이 일어나지 않기 때문이다. 이럴 때 고려해 볼 수 있는 연산자가 # 연산자이다!

문자열 내에서 매크로 매개변수 치환: # 연산자

[# 연산자를 이용해서 정의된 매크로]

```
#define      STR(ABC)      #ABC
```

매개변수 ABC에 전달되는 인자를 문자열 "ABC"로 치환해라!

[# 연산자 기반의 매크로 확장 결과]

```
STR(123)          →    "123"
STR(12, 23, 34)   →    "12, 23, 34"
```

연산자를 이용한 문제의 해결!

다음 문장 선언은

```
char * str="ABC" "DEF";
```

다음의 문장 선언과 같음과

```
char * str="ABCDEF"
```

연산자를 근거로 하여 앞서 제시한 문제를 해결한 결과

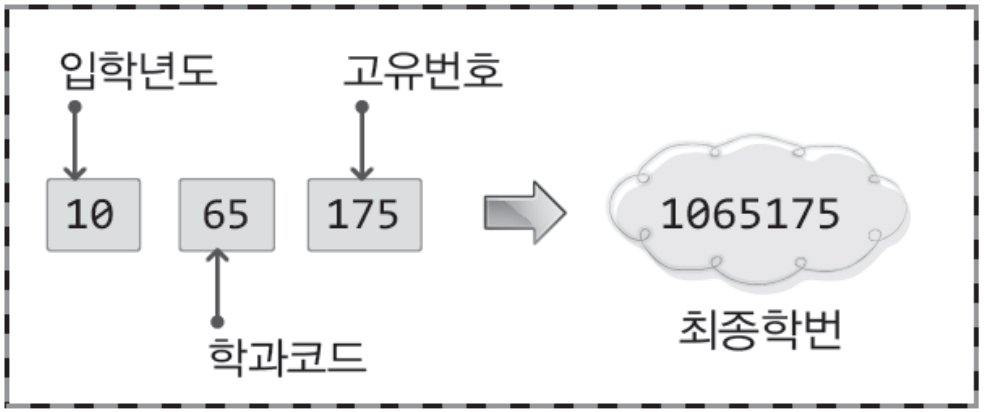
둘 이상의 문자열 선언을 나란히 하면, 이는 하나의 문자열 선언으로 인식이 된다.

```
#define STRING_JOB(A, B)   #A "의 직업은 " #B "입니다."
int main(void)
{
    printf("%s \n", STRING_JOB(이동춘, 나무꾼));
    printf("%s \n", STRING_JOB(한상순, 사냥꾼));
    return 0;
}
```

실행결과

```
이동춘의 직업은 나무꾼입니다.
한상순의 직업은 사냥꾼입니다.
```

매크로 연산자 없이 단순 연결은 불가능



세 개의 숫자를 단순 연결하여 학번을 구성해야 한다고 가정해 보자. 그리고 이를 위한 매크로를 정의해 보자.

```
#define   STNUM(Y, S, P)     YSP
```

치환 대상 YSP를 연결해 놓으면 이는 그냥 YSP로 인식이 된다.

```
#define   STNUM(Y, S, P)     Y  S  P
```

치환은 제대로 이뤄지나 Y와 S와 P가 연결되어 있지 않아서 10 65 175 와 같이 치환이 이뤄진다.

```
#define   STNUM(Y, S, P)     ((Y)*100000+(S)*1000+(P))
```

연산자를 모르는 상태에서의 최선의 해결책!

단순 연결 관련 예제 기반 문제점 확인

```
// #define STNUM(Y, S, P)  YSP
// #define STNUM(Y, S, P)  Y S P
#define STNUM(Y, S, P)   ((Y)*100000+(S)*1000+(P))

int main(void)
{
    printf("학번: %d \n", STNUM(10, 65, 175));
    printf("학번: %d \n", STNUM(10, 65, 075));
    return 0;
}
```

```
1065175
1065061
```

실행결과

실행결과를 보면 1065075로 치환이 이뤄지지 않았음을 알 수 있다. 이는 075가 8진수로 해석된 결과이다. 따라서 이 경우에는 다음과 같이 문장을 구성해야 한다.

printf("학번: %d ₩n", STNUM(10, 65, 75));

위 예제에서 보이는 매크로도 좋은 해결책이 되지 못한다. 필요한 것은 '단순 연결'인데, 8진수 인식의 문제로 인해서 생각할 요소가 발생하기 때문이다! 이는 매크로 STNUM의 사용에 있어서 주의를 요하는 요소가 되므로 좋은 매크로 정의라 할 수 없다!

필요한 형태대로 단순 결합: ## 연산자

[##연산자를 이용해서 정의된 매크로]

```
#define     CON(UPP, LOW)     UPP ## 00 ## LOW
```

매개변수 UPP와 LOW에 달되는 인자를 UPP00LOW로 단순히 연결해서 치환해라!

[## 연산자 기반의 매크로 확장 결과]

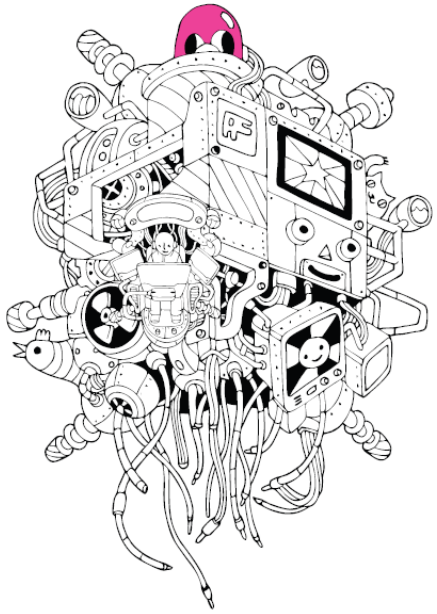
```
int num = CON(22, 77);
```

```
220077
```

이렇듯 ## 연산자를 이용해서 다음과 같이 매크로를 정의해야 학번을 단순치환 할 수 있다.

```
#define     STNUM(Y, S, P)     Y ## S ## P
```

윤성우의
열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

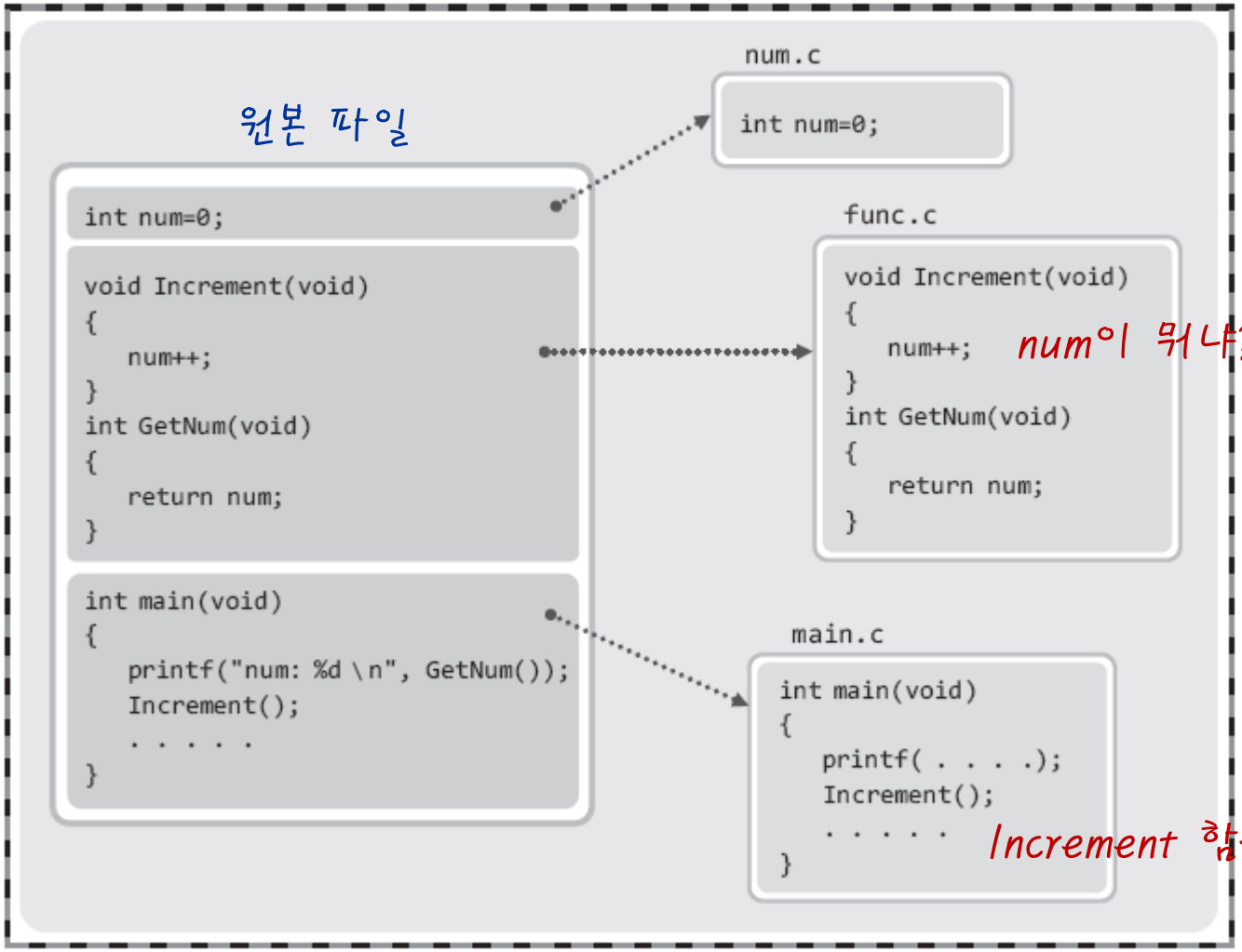
Chapter 27. 파일의 분할과 헤더파일의 디자인

윤성우의
열혈 C 프로그래밍

Chapter 27-1. 파일의 분할

윤성우 저 열혈강의 C 프로그래밍 개정판

파일을 그냥 나눠도 될까요?



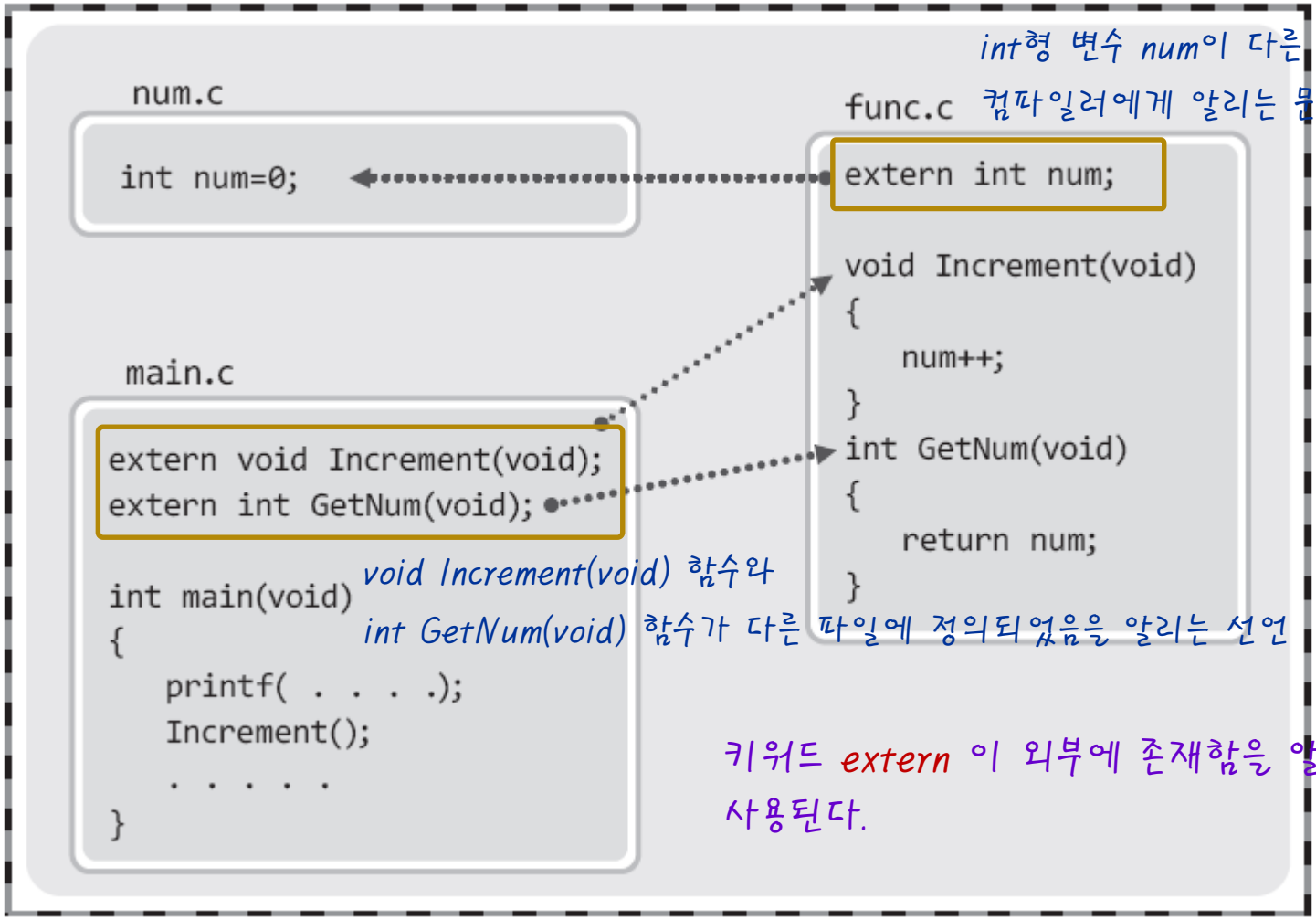
컴파일러는

파일 단위로 컴파일!

num이 뭐냐?

Increment 함수는 어디 있는 거야?

외부 선언 및 정의 사실을 컴파일러에게 알려줘야...



전역변수의 static 선언의 의미

지역변수로 선언되는 경우

```
void SimpleFunc(void)
{
    static int num=0;
    . . . .
}
```

함수 내에서만 접근이 가능한, 전역변수와 마찬가지로 한번 메모리 공간에 저장되면 종료 시까지 소멸되지 않고 유지되는 변수의 선언

전역변수로 선언되는 경우

```
static int num=0;

void SimpleFunc(void)
{
    . . . .
}
```

이 경우 int num은 전역변수이다. 단 외부 소스파일에서 접근이 불가능한 전역변수가 된다. 즉, 접근의 범위를 파일로 제한하게 된다.

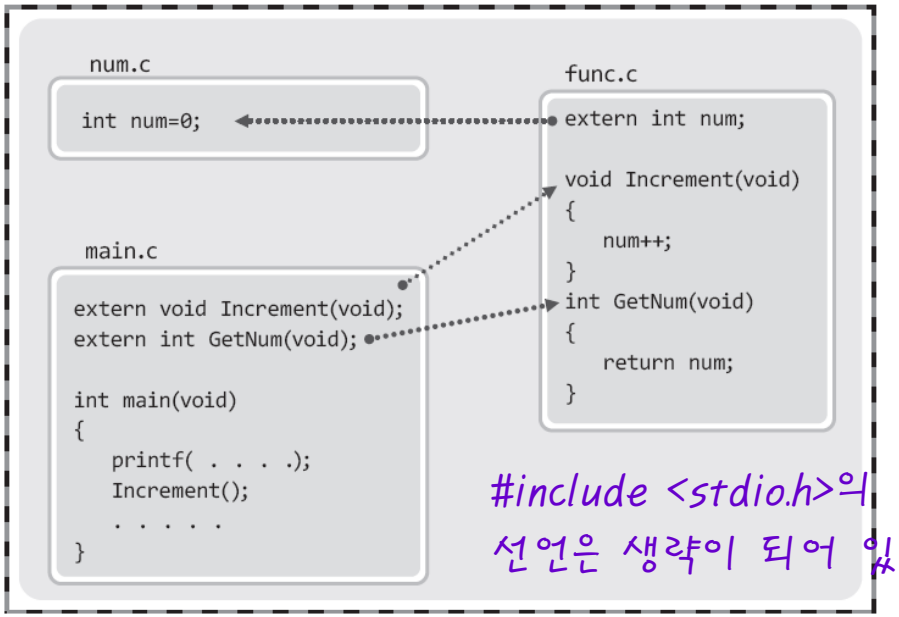
윤성우의

열혈 C 프로그래밍

Chapter 27-2. 둘 이상의 파일을 컴파일 하는 방법과 static에 대한 고찰

윤성우 저 열혈강의 C 프로그래밍 개정판

파일부터 정리하고 시작합시다.



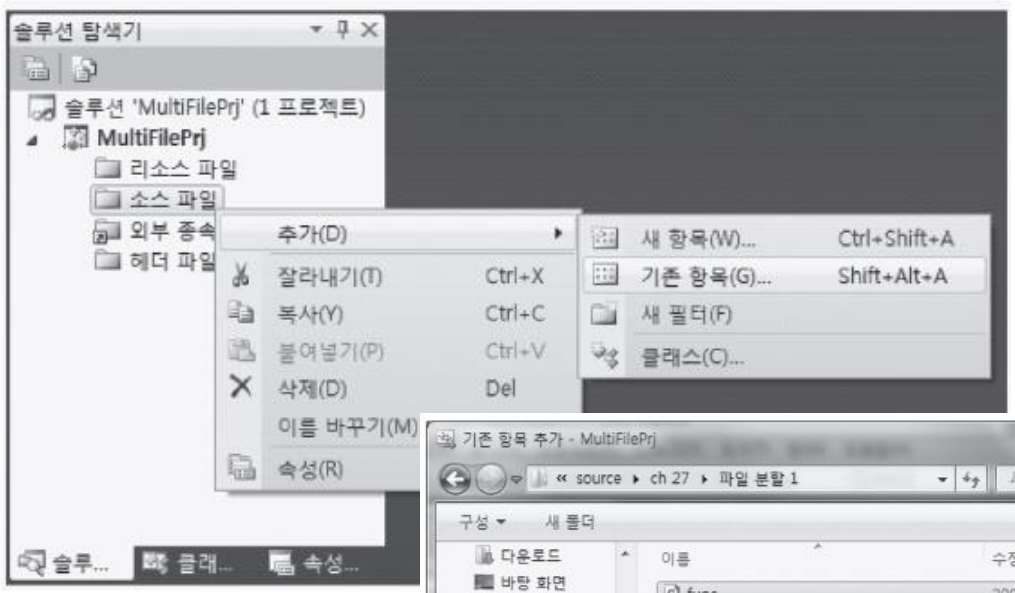
이 세 개의 파일을 하나의 프로젝트 안에 담아서 하나의 실행파일을 생성해 보는 것이 목적!

다중파일 컴파일 방법 두 가지

- 첫 번째 방법
  → 파일을 먼저 생성해서 코드를 삽입한 다음에 프로젝트에 추가한다.
- 두 번째 방법
  → 프로젝트에 파일을 추가한 다음에 코드를 삽입한다.

존재하는 파일, 프로젝트에 추가하는 방법

이미 존재하는 소스파일을 추가하는 방법



1단계

아래에서 보이듯이 다수의 파일이 하나의 프로젝트 안에 포함되었음이 솔루션 탐색기에 나타나야 한다.

2단계

추가결과 확인

프로젝트에 새로운 파일을 추가하는 방법

새로운 소스파일을 만들어서 추가하는 방법

이는 기존에 해왔던, 소스파일을 새로 생성해서 프로젝트에 추가하는 방법과 100% 동일하다.
그 과정을 재차 진행하면 새로운 소스파일을 생성해서 프로젝트 내에 포함시킬 수 있다.

함수에도 static 선언을 할 수 있습니다.

함수를 대상으로 하는 static 선언

```
static void MinCnt(void)
{
    cnt--;
}
```

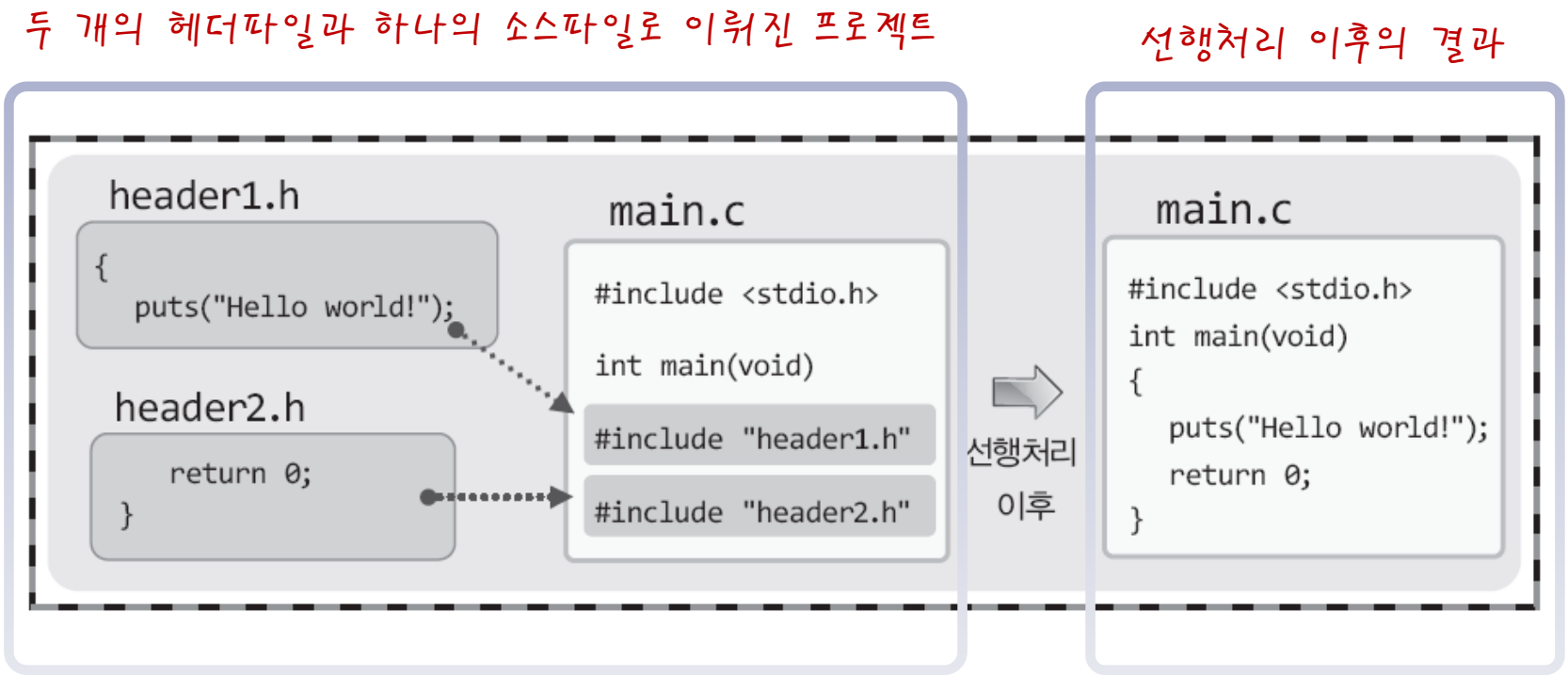
함수의 static 선언은 전역변수의 static 선언과 그 의미가 동일하다. 즉, 외부 소스파일에서의 접근을(호출을) 허용하지 않기 위한 선언이다.

윤성우의
열혈 C 프로그래밍

Chapter 27-3. 헤더파일의 디자인과 활용

윤성우 저 열혈강의 C 프로그래밍 개정판

#include 지시자와 헤더파일의 의미



위의 그림을 통해서 이해할 수 있듯이 #include 지시자는 헤더파일을 단순히 포함시키는 기능을 제공한다. 그리고 기본적으로 헤더파일에는 무엇이든 넣을 수 있다. 그러나 아무것이나 넣어서는 안 된다.

헤더파일을 include 하는 두 가지 방법

표준 헤더파일의 포함

```
#include <헤더파일 이름>
```

표준헤더 파일을 포함시킬 때 사용하는 방식이다. 표준헤더 파일이 저장된 디렉터리에서 헤더파일을 찾아서 포함을 시킨다.

프로그래머가 정의한 헤더파일의 포함

```
#include "헤더파일 이름"
```

프로그래머가 정의한 헤더파일을 포함시킬 때 사용하는 방식이다. 이 방식을 이용하면 이 문장을 포함하는 소스파일이 저장된 디렉터리에서 헤더파일을 찾게 된다.

절대경로의 지정과 그에 따른 단점

이렇듯 헤더파일의 경로를 명시할 수도 있다.

```
#include  "C:\CPower\MyProject\header.h"
```

Windows의 절대경로 지정방식.

```
#include  "/CPower/MyProject/header.h"
```

Linux의 절대경로 지정방식

▶ 절대경로를 지정하면 프로그램의 소스파일과 헤더파일을 임의의 위치로 이동시킬 수 없다(동일 운영체제를 기반으로 하더라도).

▶ 운영체제가 달라지면 디렉터리의 구조가 달라지기 때문에 경로지정에 대한 부분을 전면적으로 수정해야 한다.

상대경로의 지정 방법

상대경로 기반의 #include 선언

```
#include "header.h"
#include "Release₩header0.h"
#include "..₩CProg₩header1.h"
#include "..₩..₩MyHeader₩header2.h"
```

이 문장을 포함하는 소스파일이 저장된 디렉터리
:현재 디렉터리

현재 디렉터리의 서브인 Release 디렉터리

현재 디렉터리의 상위 디렉터리의
서브인 Cporg 디렉터리

현재 디렉터리의 상위 디렉터리의 상위
디렉터리의 서브인 MyHeader 디렉터리

위와 같은 형태로(상대경로의 지정방식을 기반으로) 헤더파일 경로를 명시하면 프로그램의 소스코드가 저장되어 있는 디렉터리를 통째로 이동하는 경우 어디서든 컴파일 및 실행이 가능해진다.

헤더파일에 무엇을 담으면 좋겠습니까?

헤더파일에 삽입이 되는 가장 일반적인 선언의 유형

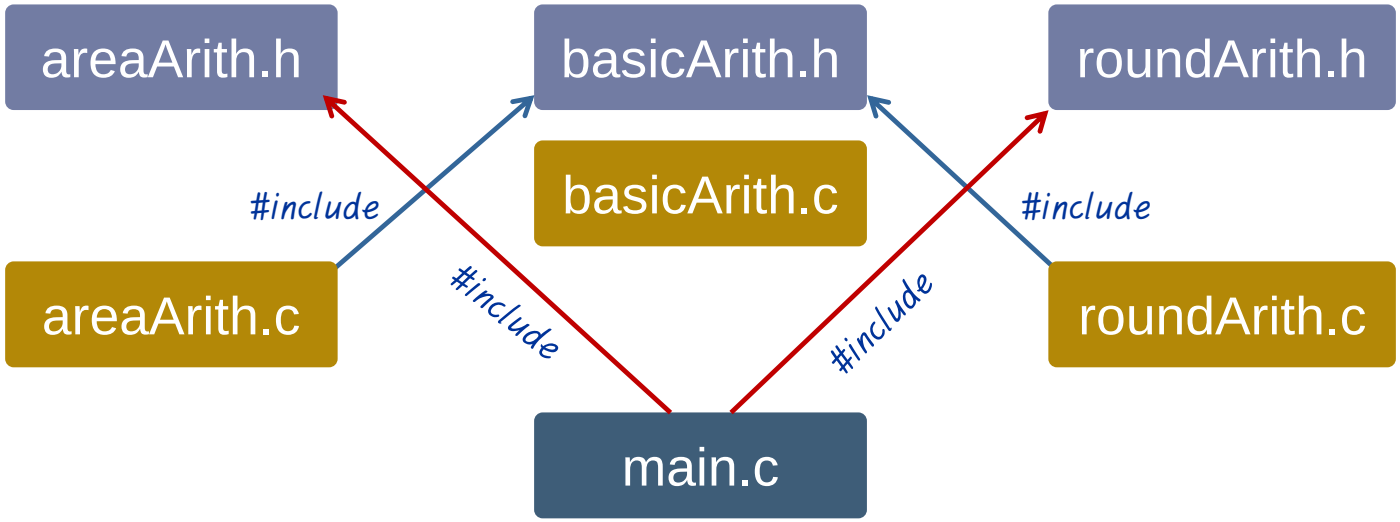
```
extern int num;
extern int GetNum(void);      // extern 생략 가능
```

- 총 7개의 소스파일과 헤더파일로 이뤄진 예제를 통해서 다음 두 가지에 대한 정보를 얻자!
  - 소스파일을 나누는 기준
  - 헤더파일을 나누는 기준 및 정의의 형태

- 예제의 소스파일과 헤더파일의 구성
  - basicArith.h basicArith.c
  - areaArith.h areaArith.c
  - roundArith.h roundArith.c
  - main.c

헤더파일과 소스파일의 포함관계

- 예제의 소스파일과 헤더파일의 구성 및 내용
  - basicArith.h basicArith.c → 수학과 관련된 기본적인 연산의 함수의 정의 및 선언
  - areaArith.h areaArith.c → 넓이계산과 관련된 함수의 정의 및 선언
  - roundArith.h roundArith.c → 둘레계산과 관련된 함수의 정의 및 선언
  - main.c



basicArith.h & basicArith.c

basicArith.h : 기본연산 함수의 선언

```
#define PI 3.1415
double Add(double num1, double num2);
double Min(double num1, double num2);
double Mul(double num1, double num2);
double Div(double num1, double num2);
```

매크로의 정의는 파일단위로 유효하다. 그래서 PI와 같은 상수의 선언은 헤더파일에 정의하고, 이를 필요한 모든 소스파일이 PI가 선언된 헤더파일을 포함하는 형태를 띤다.

basicArith.c : 기본연산 함수의 정의

```
double Add(double num1, double num2)
{
    return num1+num2;
}
double Min(double num1, double num2)
{
    return num1-num2;
}
double Mul(double num1, double num2)
{
    return num1*num2;
}
double Div(double num1, double num2)
{
    return num1/num2;
}
```

areaArith.h & areaArith.c

```
double TriangleArea(double base, double height);
double CircleArea(double rad);
```

areaArith.h : 넓이계산 함수의 선언

```
#include "basicArith.h"

double TriangleArea(double base, double height)
{
    return Div(Mul(base, height), 2);
}
double CircleArea(double rad)
{
    return Mul(Mul(rad, rad), PI);
}
```

areaArith.c : 넓이계산 함수의 정의

roundArith.h & roundArith.c

```
double RectangleRound(double base, double height);
double SquareRound(double side);
```

roundArith.h : 둥레계산 함수의 선언

```
#include "basicArith.h"

double RectangleRound(double base, double height)
{
    return Mul(Add(base, height), 2);
}
double SquareRound(double side)
{
    return Mul(side, 4);
}
```

roundArith.c : 둥레계산 함수의 정의

main.c

```c
#include <stdio.h>
#include "areaArith.h"
#include "roundArith.h"

int main(void)
{
    printf("삼각형 넓이(밑변 4, 높이 2): %g \n",
        TriangleArea(4, 2));
    printf("원 넓이(반지름 3): %g \n",
        CircleArea(3));

    printf("직사각형 둘레(밑변 2.5, 높이 5.2): %g \n",
        RectangleRound(2.5, 5.2));
    printf("정사각형 둘레(변의 길이 3): %g \n",
        SquareRound(3));
    return 0;
}
```

실행결과

```
삼각형 넓이(밑변 4, 높이 2): 4
원 넓이(반지름 3): 28.2735
직사각형 둘레(밑변 2.5, 높이 5.2): 15.4
정사각형 둘레(변의 길이 3): 12
```

구조체의 정의는 어디에? : 문제의 제시

구조체의 정의도 파일 단위로만 그 선언이 유효하다. 따라서 필요하다면 동일한 구조체의 정의를 소스파일마다 추가시켜 줘야 한다.

소스파일 intdiv.c

```
typedef struct div
{
    int quotient;     // 몫
    int remainder;    // 나머지
} Div;

Div IntDiv(int num1, int num2)
{
    Div dval;
    dval.quotient=num1/num2;
    dval.remainder=num1%num2;
    return dval;
}
```

소스파일 main.c

```
typedef struct div
{
    int quotient;     // 몫
    int remainder;    // 나머지
} Div;

extern Div IntDiv(int num1, int num2);

int main(void)
{
    Div val=IntDiv(5, 2);
    printf("몫: %d \n", val.quotient);
    printf("나머지: %d \n", val.remainder);
    return 0;
}
```

같은 구조체 정의를 둘 이상의 소스파일에 직접 추가시킨다는 것 자체가 부담!

구조체의 정의는 어디에? : 해결책의 제시

헤더파일 stdiv.h

```
typedef struct div
{
    int quotient;     // 몫
    int remainder;    // 나머지
} Div;
```

구조체의 정의도 헤더파일에 넣어두고

필요할 때마다 include 하는 것이 일반적이다!

#include

#include

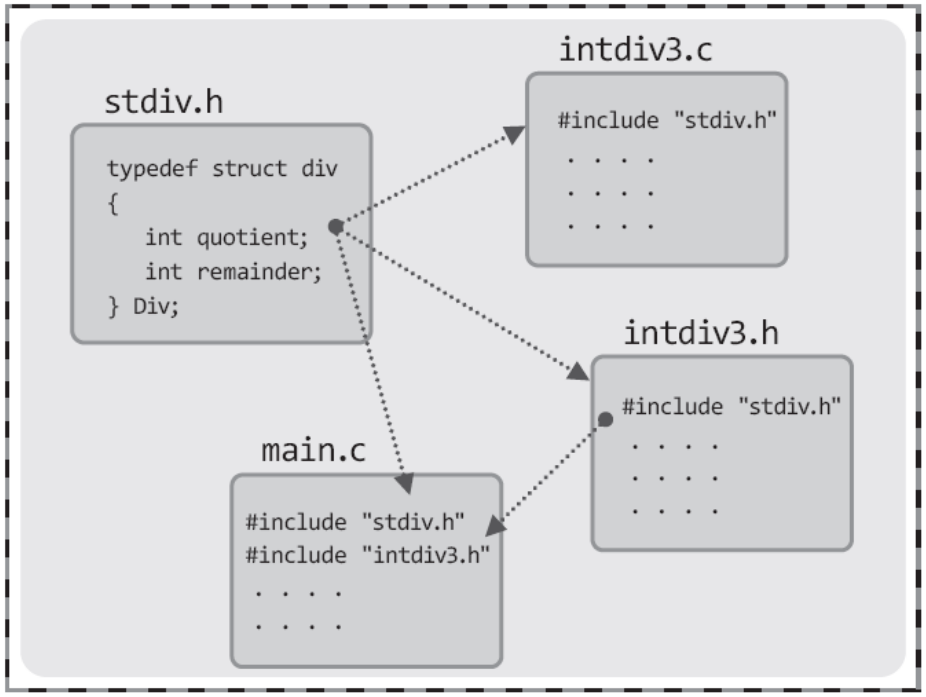
```
#include "stdiv.h"

Div IntDiv(int num1, int num2)
{
    Div dval;
    dval.quotient=num1/num2;
    dval.remainder=num1%num2;
    return dval;
}
```

소스파일 intdiv2.c

```
#include "stdiv.h"

extern Div IntDiv(int num1, int num2);

int main(void)
{
    Div val=IntDiv(5, 2);
    printf("몫: %d \n", val.quotient);
    printf("나머지: %d \n", val.remainder);
    return 0;
}
```

소스파일 main.c

헤더파일의 중복삽입 문제

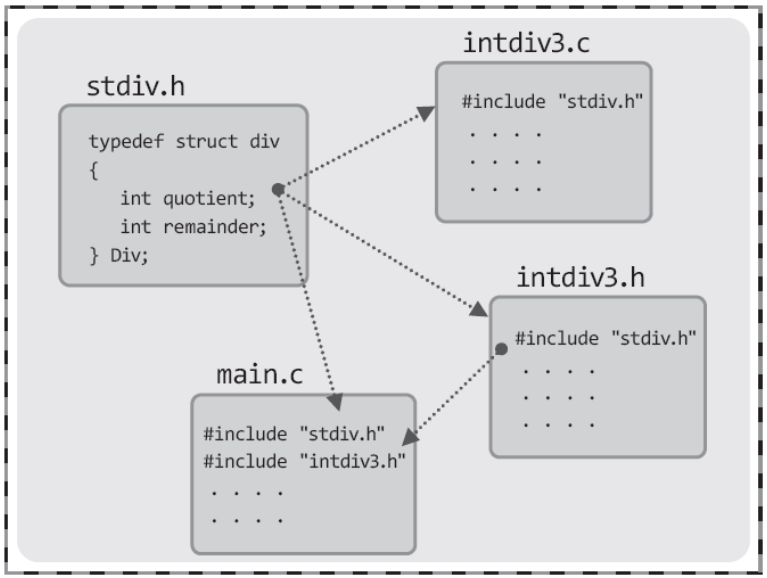


헤더파일을 직접적으로 또는 간접적으로 두 번 이상 포함하는 것 자체는 문제가 아니다. 그러나 두 번 이상 포함시킨 헤더파일의 내용에 따라서 문제가 될 수 있다.

일반적으로 선언(예로 함수의 선언)은 두 번 이상 포함시켜도 문제되지 않는다. 그러나 정의(예로 구조체 및 함수의 정의)는 두 번 이상 포함시키면 문제가 된다.

main.c는 결과적으로 구조체 Div의 정의를 두 번 포함하는 꼴이 된다! 그런데 구조체의 정의는 하나의 소스 파일 내에서 중복될 수 없다!

조건부 컴파일을 활용한 중복삽입 문제의 해결



중복 삽입문제의 해결책

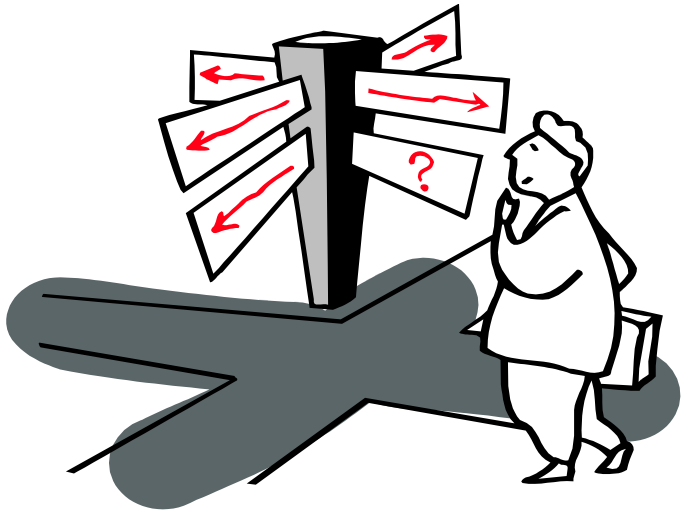
```
#ifndef __STDIV2_H__
#define __STDIV2_H__

typedef struct div
{
    int quotient;     // 몫
    int remainder;    // 나머지
} Div;

#endif
```

매크로 __STDIV2_H__ 와 #ifndef의 효과가 main.c에서
어떻게 나타나는지 그려보자!

위와 같은 이유로 모든 헤더파일은 #ifndef~#endif로 감싸는 것이 안전하고 또 일반적이다!

강의가 끝났습니다.

'윤성우의 열혈 C 프로그래밍'을 사랑해 주신 여러분께 진심으로 감사드립니다.



Chapter 27이 끝났습니다. 질문 있으신지요?