일일 업무 사항 정리

작성자	제품팀 이민성 인턴
업무 일시	20221107~20221111

세부 사항

1. 업무 내역 요약 정리

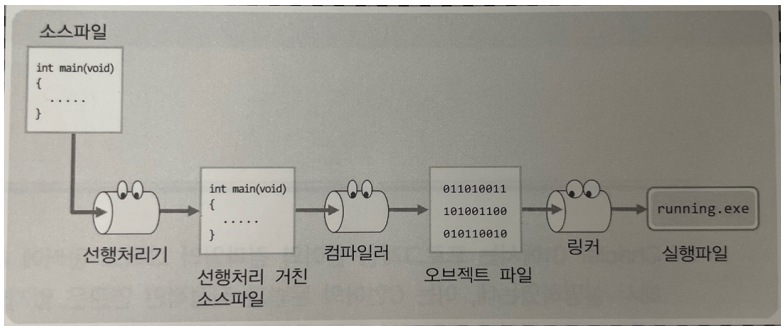
목표 내역	Done & Plan
1 주차) C 언어 개요 2 주차) 연산자/제어문 3 주차) 함수 4 주차) 배열/포인터 5 주차) 배열/포인터 다시 보기 6 주차) 구조체/공용체 7 주차) 전처리기	1. 선행처리기와 매크로 2. 대표적인 선행처리 명령문 3. 조건부 컴파일을 위한 매크로 4. 매개변수의 결합과 문자열화 5. 헤더파일의 디자인과 활용

2. 내용 세부 (업무 세부 내역 정리 및 기타 사항 정리)

1. 선행처리기와 매크로

- 선행처리는 컴파일 이전의 처리를 의미한다.

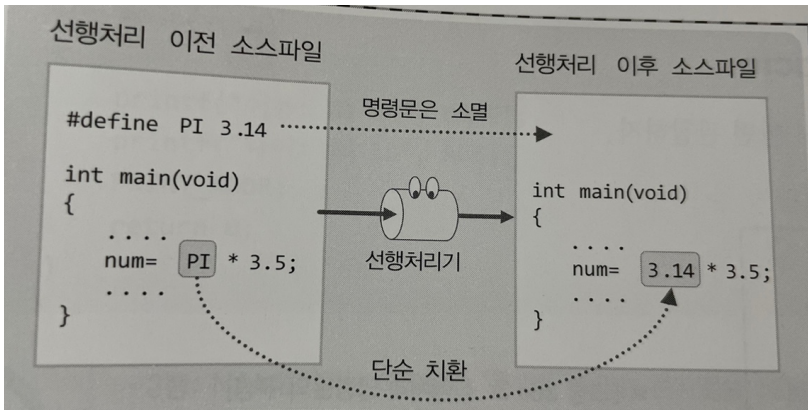
다음 그림에서 보이듯이 선행처리는 선행처리기에 의해서, 컴파일은 컴파일러에 의해서, 그리고 링크는 링커에 의해서 진행이 된다. 그런데 이 그림에서는 컴파일 이전에 선행처리의 과정을 거친다는 점에 주목을 해야 한다.



위의 그림에서 보이듯이, 컴파일 과정을 거치게 되면 바이너리 데이터로 이루어진 오브젝트 파일이 생성된다. 그렇다면 컴파일 이전에 진행되는 선행처리의 과정을 거치게 되면 어떠한 데이터로 채워진 파일이 생성될까? 선행처리의 과정을 거쳐서 생성되는 파일도 그냥 소스파일일 뿐이다. 왜냐하면 소스파일의 형태가 그대로 유지되기 때문이다. 선행처리기가 하는 일은 지극히 단순하다. 삽입해 놓은 선행처리 명령문대로 소스코드의 일부를 수정할 뿐인데, 여기서 말하는 수정이란, 단순 치환의 형태를 띠는 경우가 대부분이다. 예를 들어서,

#define PI 3.14

이처럼 선행처리 명령문은 #문자로 시작을 하며, 컴파일러가 아닌 선행처리기에 의해서 처리되는 문장이기 때문에 명령문의 끝에 세미콜론을 붙이지 않는다. 그리고 이렇게 구성이 된 명령문은 선행처리기에 "PI를 만나면 3.14 로 치환을 하라"는 메시지를 전달한다. 따라서 위의 명령문이 삽입되어 있는 소스파일은 선행처리의 과정에서 다음과 같은 방식으로 변환이 된다.



여기서 말하는 선행처리란, 컴파일 이전의 처리를 의미한다. 따라서 소스파일은 컴파일러에 의해서 컴파일 되기 이전에, 선행처리기에 의해서 선행처리의 과정을 거치게 된다고 이야기한다.

- #include <stdio.h>

자주 사용해왔던 #include <stdio.h> 선언도 #문자로 시작하는 선행처리 명령문이다. 이는 "stdio.h 파일의 내용을 이곳에 가져다 놓아라"라는 의미를 지닌다. 따라서 내가 직접 stdio.h 파일을 열어서 그 안에 있는 내용을 옮겨놓아도 같은 효과를 기대할 수 있다.

2. 대표적인 선행처리 명령문

- #define: Object-like macro

앞서 정의한 #define 명령문을 다시 한번 관찰하자

```
#define     PI      3.1415
지시자   매크로 매크로 몸체
```

위 그림에서 보이듯이 선행처리 명령문은 기본적으로 세 부분으로 나뉘는데, 제일 먼저 등장하는 #define을 가리켜 지시자라 한다. 선행처리기가 이 부분을 보고 프로그래머가 지시하는 바를 파악하지 때문에 지시자라 한다. #define 지시자 뒤에 등장하는 것을 가리켜 매크로라 하고, 그 뒤에 등장하는 것을 가리켜 매크로 몸체라 한다. 따라서 위의 선행처리 명령문은 "매크로 PI를 매크로 몸체 3.1415로 전부 치환하라"라는 것을 선행처리기에 지시한다. 결과적으로 PI라는 이름의 매크로는 그 자체로 상수 3.1415 가 된 셈이다. 참고로 PI와 같은 매크로를 가리켜 '오브젝트와 유사한 매크로' 또는 간단히 매크로 상수라 한다.

```c
#include <stdio.h>

#define NAME "홍 길 동 "
#define AGE 24
#define PRINT_ADDR puts("주 소  경 기 도  용 인 시 \n");

int main(void)
{
        printf("이 름 : %s \n", NAME);
        printf("나 이 : %d \n", AGE);
        PRINT_ADDR;
        return 0;
}
```

위 예제에서 보이는 매크로처럼 매크로의 이름은 대문자로 정의하는 것이 일반적이다. 대문자로 정의함으로써 이 식별자가 매크로라는 사실을 부각시킬 수 있다.

- #define: Function-like macro

매크로는 매개변수가 존재하는 형태로도 정의할 수 있다. 그리고 이렇게 매개변수가 존재하는 매크로는 그 동작방식이 마치 함수와 유사하여 함수와 유사한 매크로라 하는데, 줄여서 간단히 매크로 함수라 부르기도 한다. 다음은 매크로 함수의 예이다.

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

#define으로 시작을 하는 것은 매크로 상수의 정의와 동일하다. 그러나 매크로에 괄호가 등장함으로 인해서 "SQUARE(X)라는 패턴이 등장 시 X*X유형으로 바꾸라는 것"으로 해석이 된다. 여기서 괄호 안에 존재하는 X(X라는 이름이 중요한 것이 아니라, 괄호 안에 존재한다는 사실이 중요, X라는 이름은 변경 가능)는 정해지지 않은 임의의 값(또는 문장)을 의미한다. 따라서 위에서 정의한 매크로를 접한 선행처리기는 SQUARE(X)와 동일한 패턴을 만나면, 무조건 X*X로 치환해버린다. 예를 들어서 위의 매크로 정의 이후에 다음과 같은 문장을 접했다고 가정해보자.

```
SQUARE(123);
SQUARE(NUM);
```

그러면 선행처리 후에는 다음과 같이 변경이 된다.

```
123*123;
NUM*NUM;
```

```c
#include <stdio.h>
#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;
}
```

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

위 예제를 통해서 2 행에 정의된 매크로가 함수처럼 동작함을 확인하였다. 더불어 14 행의 출력결과를 통해서 2 행에 정의된 매크로에 문제가 있음도 확인했다.

- 잘못된 매크로의 정의

앞서 보인 예제의 2 행에 정의된 매크로에 어떠한 문제가 있을까?

SQUARE(3+2)

이를 함수의 관점에서 본다면 3 과 2 의 합인 5 를 SQUARE 함수의 인자로 전달하는 것으로 생각하는 것이 당연하다. 즉, 25 가 반환되어야 한다. 그러나 출력결과는 11 이다.

먼저 연산을 하고, 그 연산결과를 가지고 함수를 호출하게끔 돕는 것은 컴파일러이지 선행처리기가 아니다. 그런데 매크로는 선행처리기에 의해서 처리가 된다. 때문에 위의 문장은 단순히 다음과 같이 치환될 뿐이다.

3+2*3+2

따라서 결과로 11 이 출력된다. 해결방법은 SQUARE((3+2))로 구성하면 다음과 같이 치환되어 문제는 해결된다.

(3+2)*(3+2)

- 매크로 몸체에 괄호를 치자

#define SQUARE(X) X*X를 #define SQUARE(X) (X)*(X)로 정의하면 위에서 발생한 문제를 해결할 수 있다. (3+2)*(3+2)로 치환되기 때문이다.

그러나 여전히 문제는 남아있다. 다음 문장을 예로 들겠다.

int num = 120 / SQUARE(2);

SQUARE(2)는 4 이므로 변수 num이 30 으로 초기화될 것을 기대할 수 있다. 그런데 실제로 초기화되는 값은 120 이다. 왜냐하면 int num = 120 / (2) *(2);로 치환되어 나눗셈이 먼저 진행되지 때문이다. 따라서 이런 저런 문제를 모두 해결하기 위해서는 다음과 같은 형태로 매크로 함수를 정의해야 한다.

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

이처럼 매크로 함수를 정의할 때에는 매크로의 몸체부분을 구성하는 X와 같은 전달인자 하나하나에 괄호를 해야 함은 물론이고, 반드시 전체를 괄호로 한번 더 묶어줘야 한다.

- 매크로를 두 줄에 걸쳐서 정의하려면?

정의하는 매크로의 길이가 길어지는 경우에는 가독성을 높이기 위해서 두 줄에 걸쳐서 매크로를 정의하기도 한다. 그런데 다음과 같이 임의로 줄을 변경하면 에러가 발생한다. 기본적으로 매크로는 한 줄에 정의하는 것이 원칙이기 때문이다.

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

따라서 매크로를 두 줄 이상에 걸쳐서 정의할 때에는 다음과 같이 ₩문자를 활용해서줄이 바뀌었음을 명시해야 한다.

```
#define SQUARE ₩
((X)*(X))
```

- 매크로 정의 시, 먼저 정의된 매크로도 사용이 가능하다.

먼저 정의된 매크로는 뒤에서 매크로를 정의할 때 사용 가능하다. 다음 예제를 통해서 이를 간단히 보이겠다. 더불어서 매개변수가 두 개 이상인 경우의 매크로 함수를 예제로 제시하겠다.

```c
#include <stdio.h>
#define PI 3.14
#define PRODUCT(X, Y) ((X)*(Y))
#define CIRCLE_AREA(R) (PRODUCT((R), (R))*PI)

int main(void)
{
        double rad=2.1;
        printf("반 지 름  %g인  원 의  넓 이 : %g \n", rad, CIRCLE_AREA(rad));
        return 0;
}
```

```
반 지 름  2.1인  원 의  넓 이 : 13.8474
```

위 예제의 4 행에서는 2 행과 3 행에서 정의한 매크로를 이용해서 매크로 함수를 정의하고 있다. 이렇듯 먼저 정의된 매크로는 새로운 매크로를 정의하는데 사용이 가능하다.

- 매크로 함수의 장점

매크로 함수를 정의하는 것은 일반 함수를 정의하는 것보다 복잡하다. 그리고 정의하고자 하는 함수의 크기가 크면, 매크로로 정의하는 것 자체가 불가능할 수도 있다. 그럼에도 불구하고 함수를 매크로로 정의하는 이유는 무엇일까? 이에 대한 이해를 위해서 매크로 함수의 장점과 단점을 살펴보겠다. 다음은 함수를 매크로로 정의할 때 얻게 되는 장점들이다.

* 매크로 함수는 일반 함수에 비해 실행속도가 빠르다.
* 자료형에 따라서 별도로 함수를 정의하지 않아도 된다.

그럼 이 중에서 실행속도가 빠른 이유부터 살펴보자. 함수가 호출되면, 다음 사항들이 동반된다.

* 호출된 함수를 위한 스택 메모리의 할당
* 실행위치의 이동과 매개변수로의 인자 전달
* return 문에 의한 값의 반환

따라서 함수의 빈번한 호출은 실행속도의 저하로 이어진다. 반면 매크로 함수는 선행처리기에 의해서 매크로 함수의 몸체부분이 매크로 함수의 호출 문장을 대신하기 때문에, 위에서 언급한 사항들을 동반하지 않는다. 따라서 실행속도상의 이점이 있다.

- 매크로 함수의 단점

* 정의하기가 까다롭다.
* 디버깅하기가 쉽지 않다.

→ 그래서 작은 크기의 함수, 호출의 빈도수가 높은 함수로 정의해야한다.

3. 조건부 컴파일을 위한 매크로

매크로 지시자중에는 특정 조건에 따라 소스코드의 일부를 삽입하거나 삭제할 수 있도록 디자인된 지시자가 있다.

- #if...#endif: 참이라면

if문이 조건부 실행을 위한 것이라면, #if...#endif는 조건부 코드 삽입을 위한 지시자이다.

```
#include <stdio.h>
#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 참이라면
        printf("%d + %d = %d \n", num1, num2, num1 + num2);
#endif

#if MIN // MIN이 참이라면
        printf("%d - %d = %d \n", num1, num2, num1-num2);
#endif

        return 0;
}
```

11~13 행: #if문 뒤에는 반드시 #endif문이 등장해야 하고, 이 두 지시자 사이에 존재하는 코드는 조건에 따라서 삽입 및 삭제가 된다. 11 행의 #if문은 ADD가 참이라면의 뜻을 지닌다. 즉 ADD가 참이면 이어서 등장하는 #endif 사이에 있는 코드는 삽입이 되고, 반대로 거짓이면 삭제가 된다.

15~17 행: 11~13 행과 마찬가지로 MIN이 참이면 이어서 등장하는 #endif 사이에 있는 코드는 삽입이 되고, 거짓이면 삭제가 된다.

```
두 개의 정수 입력 : 100 20
100 + 20 = 120
```

2 행과 3 행에 정의되어 있는 매크로 ADD와 MIN이 각각 1 과 0 인 관계로 12 행은 삽입이 되지만, 16 행은 삭제가 되어 위의 실행결과를 보이게 된다. (#define ADD 0 #define MIN 1 이면 MIN이 참이 되어 15~17 행이 실행됨)

- #ifdef...#endif: 정의되었다면

이 지시자의 조합도 #if...#endif와 유사하다. #if는 매크로가 참이냐 거짓이냐를 기준으로 동작한다면, #ifdef는 매크로가 정의되었느냐, 정의되지 않았느냐를 기준으로 동작한다.

```
#include <stdio.h>
// #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;
}
```

11~13 행: 11 행의 #ifdef문은 매크로 ADD가 정의되어 있다면의 뜻을 지닌다. 따라서 정의되어 이는 ADD의 값에 상관없이, ADD라는 매크로만 정의되어 있으면, 이어서 등장하는 #endif 사이에 존재하는 소스코드는 삽입이 된다.
15~17 행: MIN이라는 매크로가 정의되어 있으면 16 행은 삽입이 된다.

```
두 개의 정수 입력 : 2 3
2 - 3 = -1
```

위 예제의 2 행과 3 행에 정의되어 있는 매크로의 값은 중요하지 않기 때문에, #define ADD #define MIN과 같이 매크로의 몸체를 생략해서 정의해도된다. 이렇게 매크로가 정의되면 소스코드에 있는 ADD와 MIN은 선행처리 과정에서 공백으로 대체가 된다.

- #ifndef...#endif: 정의되지 않았다면

사이에 있는 n이 not을 의미한다. 이 매크로는 헤더파일의 중복포함을 막기위해 주로 사용된다.

- #else의 삽입: #if, #ifdef, #ifndef에 해당

if문에 else를 추가할 수 있듯이 #if, #ifdef, #ifndef문에도 #else문을 추가할 수 있다.

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

6 행: #if는 참과 거짓을 따지는 매크로이기 때문에 이렇게 비교연산이 올 수 있다.
8 행: 6 행의 매크로가 참이면 7 행이 삽입되지만, 거짓이라면 8 행의 #else에 의해 9 행이 삽입된다.

```
매크로 상수 HIT_NUM은 현재 5입니다.
```

- #elif의 삽입: #if에만 해당

if문에 else if를 여러 번 추가할 수 있듯이, #if문에도 #elif를 여러 번 추가할 수 있다. 그리고 else if의 끝을 else로 마무리할 수 있듯이, #elif의 끝을 #else로 마무리할 수 있다.

```
#include <stdio.h>
#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 현재 5,6,7은 확실히 아닙니다.");
#endif

        return 0;
}
```

```
매크로 상수 HIT_NUM은 현재 7입니다.
```

4. 매개변수의 결합과 문자열화

- 문자열 내에서는 매크로의 매개변수 치환이 발생하지 않는다.

문자열의 구성을 위한 매크로 함수를 다음의 형태로 정의하였다.

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

그리고는 STRING_JOB(이동춘, 나무꾼)이라는 매크로 문장이 "이동춘의 직업은 나무꾼입니다."라는 문자열을 만들어낼 것을 기대하였다. 하지만 다음과 같은 이유 때문에 선행처리기는 우리의 기대대로 문자열을 만들어내지 못한다. 문자열 안에서는 매크로의 매개변수 치환이 발생하지 않는다. 이 문제를 해결하기 위해서 필요한 것이 # 연산자이다.

- 문자열 내에서 매크로의 매개변수 치환이 발생하게 만들기: # 연산자

#define STR(ABC) #ABC

위의 문장은 "매개변수 ABC에 전달되는 인자를 문자열 "ABC"로 치환하라"는 뜻이다. 이렇듯 #연산자는 치환의 결과를 문자열로 구성하는 연산자이다.

```
#include <stdio.h>
#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(한상사, 사냥꾼));
        return 0;
}
```

```
이동춘의 직업은 나무꾼입니다.
한상사의 직업은 사냥꾼입니다.
```

5. 헤더파일의 디자인과 활용

- #include 지시자의 의미를 알면 헤더파일을 완전히 이해할 수 있다.

* heder1.h

```
{
        puts("Hello World!");
```

* header2.h

```
        return 0;
}
```

* main.c

```
#include <stdio.h>

int main(void)
#include "header1.h"
#include "header2.h"
```

#include “header1.h”

이는 이 문장의 위치에다가 header1.h에 저장된 내용을 가져다 놓으라는 뜻이다. 이처럼 #include 지시자는 그 이름이 의미하듯이 파일의 내용을 단순히 포함시키는 용도로 사용된다.

```
[lms@localhost c]$ ./main
Hello World!
```

- 헤더파일을 include 하는 두 가지 방법

첫 번째 방식: #include <헤더파일 이름>

두 번째 방식: #include “헤더파일 이름”

이 둘의 유일한 차이점은 포함시킬 헤더파일의 기본 경로인데, 첫 번째 방식을 사용하면 표준 헤더파일(C의 표준에서 정의하고 있는, 기본적으로 제공되는 헤더파일)이 저장되어 있는 디렉터리에서 파일을 찾게 된다. 때문에 이 방식은 stdio.h, stdlib.h 그리고 string.h와 같은 표준 헤더파일을 포함시킬 경우에 사용된다.

반면 두 번째 방식을 사용하면, 이 문장을 포함하는 소스파일이 저장된 디렉터리에서 헤더파일을 찾는다. 때문에 프로그래머가 정의하는 헤더파일을 포함시킬 때 사용하는 방식이다. 그리고 이 방식을 사용하면 다음과 같이 헤더파일의 이름뿐만 아니라, 드라이브 명과 디렉터리 경로를 포함하는 절대경로를 명시해서 헤더파일을 지정할 수 있다. 하지만 절대경로를 지정해서 헤더파일을 선언하면 다른 컴퓨터에서 컴파일 하는 일이 매우 번거로워지기 때문에 #include문에서는 절대경로를 사용하지 않는다. 대신에 이어서 설명하는 상대경로를 사용한다.

- 상대경로의 지정 방법

드라이브 명과 디렉터리 경로를 포함하는 방식을 가리켜 절대경로라 하는 이유는 컴퓨터를 옮겨도 절대로 지정한 경로는 변경되지 않기 때문이다. 반면 상대경로는 말 그대로 상대적인 경로이다. 즉 실행하는 컴퓨터의 환경에 따라서 경로가 바뀌기 때문에 상대경로라 하는 것이다. 예를 들어서 헤더파일을 다음과 같이 포함시켰다고 가정해 보자.

#include “header.h”

이 문장은 상대적이기 때문에 이 문장을 포함하는 소스파일이 c:₩aaa에 저장되어 있다면, 이 문장의 헤더파일 검색경로도 c:₩aaa가 된다. 반면 이 문장을 포함하는 소스파일이 c:₩aaa₩bbb에 저장되어 있다면, 이 문장의 헤더파일 검색경로 역시 c:₩aaa₩bbb가 된다. 이렇듯 상대경로를 기반으로 헤더파일을 선언하면, 드라이브

명이나 디렉터리 위치에 덜 영향을 받는다. 때문에 실제로는 상대경로를 기반으로 헤더파일이 선언된다.