# 일일 업무 사항 정리

| 작성자 | 제품팀 이민성 인턴 |
|---|---|
| 업무 일시 | 20221003 ~ 20221007 |

## 세부 사항

### 1. 업무 내역 요약 정리

| 목표 내역 | Done & Plan |
|---|---|
| 1 주차) C 언어 개요<br>2 주차) 연산자/제어문<br>3 주차) 함수<br>4 주차) 배열/포인터<br>5 주차) 구조체/공용체<br>6 주차) 전처리기 | 1. while문에 의한 문장의 반복<br>2. do~while문에 의한 문장의 반복<br>3. for문에 의한 문장의 반복<br>4. 조건적 실행과 흐름의 분기<br>5. 반복문의 생략과 탈출<br>6. switch문에 의한 선택적 실행과 goto문<br>7. 함수를 정의하고 선언하기<br>8. 변수의 존재기간과 접근범위 1: 지역변수<br>9. 변수의 존재기간과 접근범위 2: 전역변수, static변수, register변수<br>10. 재귀함수<br><br>원래 예정되있었던 배열/포인터를 공부하려했지만, 배열/포인터를 공부하려다보니 함수를 알아야 이해할 수 있는 내용이 있어서 함수를 먼저 공부하였습니다.<br>다음주에는 배열/포인터를 공부하겠습니다. |

**2. 내용 세부 (업무 세부 내역 정리 및 기타 사항 정리)**

# 함수

**1. while문에 의한 문장의 반복**

- 반복문이란 무엇인가?

문자열 "Hello World!"를 총 열 번 출력하고 싶다. 어떻게 하면 좋을까? 지금까지 공부했던 방법만 가지고 이야기하면, "Hello World!"를 출력하는 printf 함수 호출문을 열 번 삽입해야 한다. 그러나 이 방법은 비효율적이다. 출력의 양에 비례해서 프로그램 코드의 양도 함께 늘어나기 때문이다. 이를 해결하기 위해서 사용하는 것이 반복문이다. C언어는 while문, do~while문, for문 총 3 가지의 반복문을 제공하고 있다.

- while문은 반복을 명령하는 문장이다.

while문은 특정조건을 주고 그 조건을 만족하는 동안, 특정영역을 계속해서 반복하는 구조이다. 이와 관련해서 다음 코드를 보자.

```
#include <stdio.h>

while(num<3) // 반복의 조건은 num<3
{
        printf("Hello world! \n");
        num++;
}
```

이것이 while문이다. 이렇듯 while문의 소괄호 안에는 반복의 조건을 명시하고, 이 조건이 만족되는 동안 중괄호 안에 존재하는 코드가 반복 실행되는 구조이다. 그럼 현재 num의 값이 0 인 상태에서 위의 while문을 실행한다고 가정해보자. 제일 먼저 다음의 내용을 확인하게 된다. "num이 3 보다 작은가?" 그런데 num이 0 이라 하였으니, 당연히 3 보다 작은 상태이다. 따라서 다음 두 문장을 실행하게 된다.

printf("Hello world! ₩n");

num++ // 이로 인해 num의 값은 1 이 됨

실행이 완료되면, 다시 반복의 조건을 확인하러 올라간다. 그런데 아직도 num의 값은 1 이므로 반복의 조건을 만족한다. 따라서 다시 중괄호 안에 존재하는 코드를 실행한다. 이렇게 조건이 만족되어 중괄호 안의 코드를 반복 실행하다 보면, 언젠가는 num의 값이 3 이 되어, 반복의 조건을 만족하지 않는 상황이 된다. 그리고 이 때 while문을 벗어나, 그 다음 행을 실행하게 된다. 그럼 지금까지 설명한 내용을 실제 예제를 통해서 확인해보자.

```
#include <stdio.h>

int main(void)
{
        int num = 0;

        while(num < 5)
        {
                printf("Hello world! %d \n", num);
                num++;
        }
        return 0;
}
```

```
Hello world! 0
Hello world! 1
Hello world! 2
Hello world! 3
Hello world! 4
```

위 예제 7 행의 반복조건은 num<5 이다. 그런데 num의 값이 0 에서부터 시작했으므로 9 행과 10 행을 총 5 회 반복하게 된다. 그렇다면 이번에는 10 행을 주석처리 한 상태에서 다시 실행해보자. 그 결과가 어떻게 되는가? 프로그램이 종료하지 않고 계속해서 실행이 된다. 10 행은 7 행의 반복조건을 무너뜨리기 위한 연산이다. 따라서 10 행을 주석처리하면, 반복의 조건이 무너지지 않아서, while문의 반복이 멈추지 않는 무한루프라 불리는 형상이 만들어진다. 따라서 반복문의 구성에 있어서 중요한 것 중 하나는 **반복의 조건을 무너뜨리기 위한 최소한의 연산**이다.

- 반복문 안에서도 들여쓰기 합니다.

들여쓰기란 코드를 쉽게 구분 및 분석할 수 있도록 종속관계에 따라서 몇 칸 띄운 다음에 코드를 입력하는 것을 뜻한다.

- 반복의 대상이 하나의 문장이라면 중괄호는 생략한다.

while문에서 반복의 대상을 지정할 때 사용하는 중괄호는 반복의 대상이 하나의 문장으로 이뤄진 경우에 생략이 가능하다. 즉 중괄호는 반복의 대상이 둘 이상의 문장으로 이뤄진 경우, 이를 묶기 위해서 사용하는 것이다. 따라서 아래와 같이 줄여서 쓸 수 있다.

```
while(num < 5)
        printf("Hello world! %d \n", num++);

while(num<5)
        printf("Hello World! %d \n", num), num++;
```

- 구구단

```
#include <stdio.h>

int main(void)
{
        int dan=0, num=1;
        printf("몇 단 ? : ");
        scanf("%d", &dan);

        while(num<10)
        {
                printf("%d x %d = %d \n", dan, num, dan*num);
                num++;
        }
        return 0;
}
```

```
몇 단 ? : 5
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
```

- 무한루프의 구성

반복조건이 와야 할 위치에 0 이 아닌 값(참을 의미하는 값)이 올 경우, 소위 말해서 무한루프라 불리는 빠져나가지 않는 반복문이 형성된다. 이와 관련해서 다음 코드를 보자.

```
while(1)
{
        printf("%dx%d=%d \n", dan, num, dan*num
        num++;
}
```

숫자 1 은 참을 의미하는 대표적인 값이다. 따라서 이 값이 반복의 조건을 대신하면, 그 결과는 항상 참이 되어 빠져나가지 못하는 반복문이 구성된다. 물론 break문을 이용하면 빠져나갈 수 있으며, 실제로 특정 기능을 완성하기 위해서 break문과 함께 위의 형태로 무한루프를 구성하기도 한다.

- while문의 중첩

'while문의 중첩'은 while문 안에 while문이 포함된 상황을 뜻한다. 따라서 while문의 중첩은 새로운 문법의 이해를 요구하지 않는다. 오히려 while문에 대한 여러분의 이해를 확장시킬 것을 요구한다. 처음 접하면 이해하는데 시간이 걸릴 수 있는 while문의 중첩은 매우 흔히 사용된다. 대표적인 예가 '구구단의 전체 출력'이다. 2 단부터 9 단까지 출력을 하는데 반복문을 중첩시키지 않는다면, while문을 총 8 회 삽입해야한다. 그러나 while문을 중첩시키면 다음 예제에서 보이는 바와 같이 중첩된 while문 하나로 이 모든 것을 대신할 수 있다.

```
#include <stdio.h>

int main(void)
{
        int cur=2;
        int is=0;

        while(cur<10) // 2단부터 9단까지 반복
        {
                is=1; // 새로운 단의 시작을 위해서
                while(is<10) // 각 단의 1부터 9의 곱을 표현
                {
                        printf("%dX%d=%d \n", cur, is, cur*is);
                        is++;
                }
                cur++; // 다음 단으로 넘어가기 위한 증가
        }
        return 0;
}
```

```
2X1=2
2X2=4
2X3=6
2X4=8
2X5=10
2X6=12
2X7=14
2X8=16
2X9=18
3X1=3
3X2=6
3X3=9
3X4=12
3X5=15
3X6=18
3X7=21
3X8=24
3X9=27
4X1=4
4X2=8
```

위 예제의 8 행에 존재하는 while문은 2 단부터 9 단까지의 출력을 위한 것이다. 그리고 11 행의 while문은 각 단에서 1 부터 9 까지의 곱의 결과를 출력하기 위한 것이다. 따라서 다음의 형태로 실행이 이어진다.

"cur이 2일 때, is가 1부터 9까지 1씩 증가시키면서 13 행을 실행하게 된다 (2 단 출력)"

"cur이 3일 때, is가 1부터 9까지 1씩 증가시키면서 13 행을 실행하게 된다 (3 단 출력)"

.

.

"cur이 9일 때, is가 1부터 9까지 1씩 증가시키면서 13 행을 실행하게 된다 (9 단 출력)"

이렇듯 중첩된 반복문이 어떠한 형태로 실행되는지 그 구조를 명확히 이해하고 또 설명할 수 있다.

**2. do~while문에 의한 문장의 반복**

이번에 소개하는 do~while문도 while문과 마찬가지로 반복문이다. 그리고 이 둘의 유일한 차이점은 '반복의 조건을 검사하는 시점'에 있다.

- do~while문의 기본구성

do~while은 while문과 달리 '반복조건'을 뒷부분에서 검사한다. 이렇듯 '반복조건'을 뒷부분에서 검사하기 때문에, 처음부터 '반복조건'을 만족하지 못하면 '반복영역'을 한번도 실행하지 않는 while문과 달리, **반복영역을 최소한 한번을 실행하는 구조**이다.

```
do
{
	printf("Hello World! ₩n");
	num++;
} while(num<3);
```

위의 코드는 do~while문의 기본구성을 보이고 있다. Do 다음에 등장하는 중괄호가 반복영역이고, 중괄호에 이어서 등장한 것이 '반복조건'이다. 따라서 위의 코드는 다음 실행흐름을 갖는다.

1 단계 : 실행
2 단계 : 반복여부 확인 후, 재실행을 위해 이동
3 단계 : 다시 실행
4 단계 : 반복여부 확인 후, 재실행을 위해 이동
5 단계 : 다시 실행
6 단계 : 반복여부 확인 후, do~while문 탈출

**"while문과는 반복조건의 검사위치가 다르며, 이로 인해서 반복영역을 최소한 1 회 이상 실행한다는 점이 while문과의 유일한 차이점이다."**

- do~while문이 자연스러운 상황

이렇듯 do~while문과 while문은 매우 유사하다. 때문에 while문은 do~while문으로, 그리고 do~while문은 while문으로 바꿀 수 있다. 따라서 앞서 소개한 예제 구구단의 다음 while문은,

```
while(num<10)
{
	print("%dX%d=%d ₩n", dan, num, dan*num);
	num++;
}
```

다음과 같이 do~while문으로 어렵지 않게 바꿀 수 있다.

```
do
{
	printf("%dX%d=%d ₩n", dan, num, dan*num);
	num++;
} while(num<10);
```

그럼 언제 while문을 쓰고, 언제 do~while문을 쓸까?

사실 대부분의 경우 do~while문이 while문을 대체할 수 있다. 하지만 일반적인 경우라면 while문을 선택한다. while문의 경우, 반복의 조건이 앞 부분에서 위치해서 코드를 작성하기에도, 이해하기에도 용이하기 때문이다. 하지만 do~while문이 더 자연스러운 경우도 있다.

"반복영역이 무조건 한 번 이상 실행되어야 합니다!"
그럼 이에 해당하는 예제 하나를 제시하겠다. 다음 예제에서는 프로그램 사용자가 입력하는 정수를 계속해서 더해나간다. 그리고 이러한 덧셈은 0 이 입력될 때까지 계속된다. 즉, 0 이 입력되면 덧셈결과를 출력하고 프로그램은 종료된다.

```
#include <stdio.h>

int main(void)
{
        int total=0, num=0;

        do
        {
                printf("정 수  입 력 (0 to quit): ");
                scanf("%d", &num);
                total += num;
        } while(num!=0);
        printf("합 계 : %d \n", total);
        return 0;
}
```

```
정 수  입 력 (0 to quit): 3
정 수  입 력 (0 to quit): 4
정 수  입 력 (0 to quit): 0
합 계 : 7
```

위 예제의 경우, 프로그램 사용자로부터 하나의 정수를 입력 받고 난 다음에야 비로소 계속해서 진행할것인지 말것인지를 결정짓게 된다. 따라서 반복영역을 최소한 한번은 실행해야 한다. 때문에 이러한 경우에는 do~while문을 사용하는 것이 while문을 사용하는 것 보다 자연스럽다.

**3. for문에 의한 문장의 반복**

이번에는 반복문 중에서 가장 많이 사용되는 for문에 대해서 공부할 것이다. for문은 while문이나 do~while문과 달리 반복을 위한 변수의 선언과 반복조건을 거짓으로 만들기 위한 값의 증가 및 감소 연산등을 한데 묶을 수 있도록(쉽게 말해서 반복을 구성하기 위해 필요한 모든 것은 한데 묶을 수 있도록)만들어진 반복문이다. 따라서 새로운 형태의 반복문이라기 보다, 특정상황에서 while문이나 do~while문보다 편하게 반복문을 구성할 수 있도록 설계된 것이 for문이라 할 수 있다.

- for문의 구조와 이해

다음의 요구사항을 만족하는 while문을 작성해보자
"문자열 'Hi~'를 총 3 회 출력하고 싶습니다"
위의 문장에서 중요한 사실은 반복의 횟수가 정해져 있다는 사실이다. 이렇듯 반복의 횟수를 정한 상태에서의 While문은 다음과 같이 구성하게 된다.

```
int main(void)
{
        int num=0; // 필수요소 1. 반복을 위한 변수의 선언
        while(num<3) // 필수요소 2. 반복의 조건검사
        {
                printf('Hi~')
                num++; // 필수요소 3. 반복의 조건을 거짓으로 만들기 위한 연산
        }
        ....
```

```
}
```

정해진 횟수의 반복문을 구성하려면, 위의 코드에서 주석을 통해 표현한 필수요소 1, 2, 3 중 어느 것 하나라도 생략할 수 없다. 실제로 다음의 질문에 답을 하려면 이 세 문장을 봐야 답변이 가능하다.

"이 반복문은 몇 회 반복을 하나요?"

때문에 반복문의 분석을 위해서는 이 세 문장을 늘 관찰하기 마련이다. 그리고 이러한 이유로 다음과 같은 생각을 하지 않을 수 없다.

"필수요소 1, 2, 3 을 한 데 묶을 순 없을까?"

실제로 이 세가지 요소를 한 데 묶어 놓으면, 반복문의 반복횟수를 판단하기가 한결 수월해진다. 그리고 이러한 생각을 반영해서 만든 것이 for문이다. 그럼 for문에 대한 설명을 시작하겠다. 위의 코드에서 언급한 필수요소 1,2,3 을 가리켜 각각 다음과 같이 표현한다.

- 필수요소 1 | 초기식 | 반복을 위한 변수의 선언 및 초기화에 사용
- 필수요소 2 | 조건식 | 반복의 조건을 검사하는 목적으로 선언됨
- 필수요소 3 | 증감식 | 반복의 조건을 '거짓;으로 만드는 증가 및 감소 연산

그리고 for문을 이용하면 다음과 같이 위의 3 요소를 한 데 묶을 수 있다.

```
for(초기식; 조건식; 증감식;)
{
 // 반복의 대상이 되는 문장들
}
```

따라서 앞서 보인 while문은 다음과 같이 변경이 가능하다.

```
int main(void)
{
        for(int num=0; num<3; num++)
                printf('Hi~'); // 반복의 대상이 한 줄이므로 중괄호 생략!
        ....
}
```

비교해보면 반복문이 훨씬 간결해졌음을 알 수 있을 것이다. 그런데 아직도 일부 C컴파일러는 for문의 초기식 위치에서의 변수선언을 허용하지 않고 있다. 따라서 다음과 같이 작성을 해야 모든 C컴파일러에서 컴파일이 가능하다.

```
int main(void)
{
        int num;
        for(num=0; num<3; num++)
                printf('Hi~'); // 반복의 대상이 한 줄이므로 중괄호 생략!
        ....
}
```

for문의 실행방식은 구조적으로 while문의 실행방식과 동일하다. 하지만 for문이 항상 while문보다 좋지는 않다. for문처럼 틀이 정해져 있다는 것은 모든 경우에 있어서 최선이 될 수 없다. for문은 while문에 비해서 유연한 느낌을 주지 못한다. 반복의 횟수가 정해진 경우라면 분명 for문이 최선이다. 하지만 반복의 횟수가 딱 정해지지 않은 경우라면, 예를 들어서 프로그램 사용자가 임의의 값을 입력하기만 기다리는 상황이라면, for문보다는 while문이 더 자연스러울 수 있다.

-for문 기반의 다양한 예제

0 이상, 그리고 프로그램 사용자가 입력한 정수 이하의 값을 모두 더해서 그 결과를 출력하는 예제를 소개하겠다.

```
#include <stdio.h>

int main(void)
{
        int total=0;
        int i, num;
        printf("0부터 num까지의 덧셈, num은? ");
        scanf("%d", &num);

        for(i=0; i<num+1; i++)
                total+=i;

        printf("0부터 %d까지의 덧셈결과 : %d \n", num, total);
        return 0;
}
```

```
0부터 num까지의 덧셈, num은? 3
0부터 3까지의 덧셈결과 : 6
```

이어서 다음 예제를 소개하겠다. 이 예제에서는 프로그램 사용자가 입력하는 실수의 평균값을 출력한다. 실수의 입력은 0 보다 작은 음의 실수 값이 입력될 때까지 계속되며, 입력을 마무리하기 위해서 마지막으로 입력된 음의 실수는 평균값 계산에서 제외시켰다.

```
#include <stdio.h>

int main(void)
{
        double total=0.0;
        double input=0.0;
        int num=0;

        for( ; input>=0.0 ; )
        {
                total+=input;
                printf("실수 입력 (minus to quit) : ");
                scanf("%lf", &input);
                num++;
        }
        printf("평균 : %f \n", total/(num-1));
        return 0;
}
```

```
실수 입력 (minus to quit) : 3.3
실수 입력 (minus to quit) : 2.2
실수 입력 (minus to quit) : 1.1
실수 입력 (minus to quit) : -1.0
평균 : 2.200000
```

위 예제의 9 행에 있는 for문에는 초기식과 증감식이 없는 것을 알 수 있다. 이렇듯 불필요하다면 비워도 된다. for문은 기본적으로 초기식, 조건식, 증감식을 채울 수 있지만, 필요 없는 부분이 있다면 채우지 않아도 된다.

```
for(;;)
{
        ....
}
```

위의 코드와 같이 for문의 중간에 위치한 조건식이 비워지면 무조건 참으로 인식이 되어 무한루프를 형성하게 된다.

- for문의 중첩

```
#include <stdio.h>

int main(void)
{
        int cur, is;

        for(cur=2; cur<10; cur++)
        {
                for(is=1; is<10; is++)
                        printf("%dx%d=%d \n", cur, is, cur*is);
                printf("\n");
        }
        return 0;
}
```

위 예제를 실행하면 실제로 구구단 전체가 출력된다.

**4. 조건적 실행과 흐름의 분기**

- 흐름의 분기가 필요한 이유

사칙연산이 가능한 계산기 프로그램을 구현한다고 가정해보자. 물론 이 프로그램은 사칙연산이 모두 가능해야 한다. 단, 선택적 실행이 가능해야 한다. 예를 들어서 이 프로그램을 사용하는 사람은 자신의 필요에 따라서 덧셈 혹은 곱셈을 선택적으로 실행할 수 있어야 한다. 선택적 실행이 가능한 계산기 프로그램의 구현을 위해서는 프로그램의 흐름을 분기시킬 수 있어야 한다.

- if문을 이용한 조건적 실행

분기의 가장 기본은, 두 개의 키워드 if와 else로 구성이 되는 if~else문이다. 그런데 이 중 첫번째 키워드인 if는 독립적으로 사용되어 조건적 실행을 가능하게 한다. 이러한 if문은 다음과 같이 구성한다.

```
if(num1>num2) // num1 이 num2 보다 크다면 아래의 중괄호를 실행한다.
{
        printf('num1 이 num2 보다 큽니다. ₩n')
        printf('%d > %d ₩n', num1, num2);
}
```

위의 문장에서 num1 이 num2 보다 큰 경우에만 if문의 중괄호에 삽입된 두 개의 Printf함수 호출문이 실행된다. 물론 조건이 만족될 때 실행할 문장이 하나라면 중괄호는 생략이 가능하다. 그럼 다음 예제를 통해서 if문이 동작하는 방식을 조금 더 구체적으로 확인하도록 하자.

```
#include <stdio.h>

int main(void)
{
        int num;
        printf("정수 입력 : ");
        scanf("%d", &num);

        if(num<0) // num이 0보다 작으면 아래의 문장 실행
                printf("입력 값은 0보다 작다. \n");

        if(num>0) // num이 0보다 크면 아래의 문장 실행
                printf("입력 값은 0보다 크다. \n");

        if(num==0) // num이 0이면 아래의 문장 실행
                printf("입력 값은 0이다. \n");

        return 0;
}
정수 입력 : 5
입력 값은 0보다 크다.
```

이제 앞서 언급한 사칙연산이 선택적으로 실행가능한 계산기 프로그램을 작성해보고자 한다.

```
#include <stdio.h>

int main(void)
{
        int opt;
        double num1, num2;
        double result;

        printf("1.덧셈 2.뺄셈 3.곱셈 4.나눗셈 \n");
        printf("선택? ");
        scanf("%d", &opt);
        printf("두 개의 실수 입력 : ");
        scanf("%lf %lf", &num1, &num2);

        if(opt==1)
                result = num1 + num2;
        if(opt==2)
                result = num1 - num2;
        if(opt==3)
                result = num1 * num2;
        if(opt==4)
                result = num1 / num2;

        printf("결과 : %f \n", result);
        return 0;
}
```

```
1.덧셈  2.뺄셈  3.곱셈  4.나눗셈
선택? 2
두 개의 실수 입력 : 3 9
결과 : -6.000000
```

위의 예제는 어렵지 않다. 그러나 이 예제에는 한가지 문제점이 존재하는데, 그 문제점은 다음과 같다.
"어떠한 숫자가 입력되든 15,17,19,21 행의 조건검사(비교연산)은 모두 실행된다"
이것이 왜 문제인가? 예제에 존재하는 4 개의 if문 중 하나만이 조건을 만족하게 되어있기 때문이다. 따라서 다음과 같이 구현 가능하다면 불필요한 연산의 수를 줄일 수 있어서 더 효율적이라 할 수 있다.
"조건을 만족하는 if문이 발견되면 나머지 if문은 건너뛴다"
그리고 이러한 형태의 연산을 위해서 필요한 것이 바로 If~else문이다. if~else문은 후에 공부하고 if문과 관련된 예제를 공부하겠다. 이 예제에서는 1 이상 100 미만의 정수 중에서 3 의 배수이거나 4 의 배수인 정수를 순서대로 출력한다.

```
#include <stdio.h>

int main(void)
{
        int num;

        for(num=1; num<100; num++)
        {
                if(num%3==0 || num%4==0)
                        printf("3 또는  4의  배수 : %d \n" num);
        }
        return 0;
}
```

```
3 또는  4의  배수 : 3
3 또는  4의  배수 : 4
3 또는  4의  배수 : 6
3 또는  4의  배수 : 8
3 또는  4의  배수 : 9
3 또는  4의  배수 : 12
3 또는  4의  배수 : 15
3 또는  4의  배수 : 16
```

위 예제 9 행의 if문에는 다음의 연산식이 존재한다.
if(num%3 ==0 || num%4 ==0)
이 중에서 || 연산자보다 관계 연산자인 ==연산자의 우선순위가 높기 때문에 이는 다음과 같이 다시 쓸 수 있다.
if( (num%3 ==0) || (num%4 ==0) )
즉, 두 개의 == 연산의 결과를 대상으로 || 연산을 진행하게 된다.

- if~else문을 이용한 흐름의 분기

키워드 else는 독립적으로 사용되는 것이 아니라 키워드 if와 더불어 하나의 문장을 구성하는 형태로 사용된다.

```
if(num1>num2) // num1>num2 이 참이면 아래의 If블록을 실행
{       // if 블록
        printf("num1 이 num2 보다 크다. ₩n");
        printf("%d > %d ₩n", num1, num2);
}
else // num1>num2 이 '거짓'이면 아래의 else블록을 실행
{       // else 블록
        printf("num1 이 num2 보다 크지 않습니다. ₩n");
        printf("%d <= %d ₩n", num1, num2);
}
```

즉, if절의 조건이 참이면 중괄호로 묶여있는 if블록이 실행되고, 조건이 참이 아니면 중괄호로 묶여있는 else 블록이 실행되는 구조이다. 그럼 다음 예제를 통해서 이를 확인해보자.

```
#include <stdio.h>

int main(void)
{
        int num;
        printf("정 수  입 력 : ");
        scanf("%d", &num);

        if(num<0)
                printf("입 력  값 은  0보 다  작 다 . \n");
        else
                printf("입 력  값 은  0보 다  작 지  않 다 . \n");

        return 0;
}
```

```
정 수  입 력 : 5
입 력  값 은  0보 다  작 지  않 다 .
```

9 행의 조건 num<0 이 참이면 10 행이 실행되고, 거짓이면 12 행이 실행되는 매우 단순한 예제이다.

- if~else if~else의 구성

if~else문의 경우는 두 개의 블록 중 하나를 선택해서 실행하는 구조였는데, 이번에 설명하는 if~else if~else문은 셋 이상의 블록 중 하나를 선택해서 실행하는 구조이다. 중간에 얼마든지 else if절을 추가할 수 있다. 그리고 무엇보다 중요한 것은 하나라도 조건이 만족되어 해당 블록을 실행하고 나면, 마지막에 있는 else까지도 그냥 건너뛴다는 사실이다. 조건의 만족여부 검사는 위에서 아래로 진행이 되며, 조건이 만족되어서 해당 블록을 실행하고 나면 마지막 else까지도 건너뛴다. 이러한 특징 때문에, 특히 조건이 만족되고 해당 블록을 실행하고 나면 마지막 else까지도 그냥 건너뛴다는 특징 때문에, 앞서 소개한 계산기 예제를 효율적으로 실행되도록 다음과 같이 개선시킬 수 있다.

```
#include <stdio.h>

int main(void)
{
        int opt;
        double num1, num2;
        double result;

        printf("1.덧셈 2.뺄셈 3.곱셈 4.나눗셈 \n");
        printf("선택?");
        scanf("%d", &opt);
        printf("두 개의 실수 입력 : ");
        scanf("%lf %lf", &num1, &num2);

        if(opt==1)
                result = num1 + num2;
        else if(opt==2)
                result = num1 - num2;
        else if(opt==3)
                result = num1 * num2;
        else
                result = num1 / num2;

        printf("결과 : %f \n", result);
        return 0;
}
```

위 예제의 실행결과는 이전의 계산기 예제와 차이가 없으나, 조건이 만족되면 else이후로 건너뛰는 특징 때문에 연산의 수는 확실히 줄어든다.

- if~else if~else의 진실

if~else if~else는 if~else를 중첩시킨 형태에 지나지 않는다. if else if else는 if~else를 중첩시키되 else블록을 대상으로 중첩시킨 결과이다. 그래서 if문에 명시된 하나의 조건이 참이 되면, 나머지 전부를 건너뛰었던 것이다.

- 조건 연산자: 피 연산자가 세 개인 삼 항 연산자

if~else문을 일부 대체할 수 있는 조건 연산자에 대해서 소개하겠다. 이 연산자는 피연산자의 수가 3 개이기 때문에 삼 항 연산자로도 불린다. 이러한 조건 연산자는 다음과 같이 구성이 된다.

(num1>num2) ? (num1) : (num2);

조건 연산자는 기호 ? 와 : 으로 이뤄진다. 이렇듯 두 개의 기호가 서로 떨어져서 하나의 연산자를 구성하기 때문에 피연산자를 3 개까지 둘 수 있는 것이다. 위의 조건 연산자 문장을 다음과 같이 다시 쓰겠다.

(조건) ? data1 : data2

위의 문장에서 조건이 참이면 연산결과로 data1 이 반환되고, 조건이 거짓이면 연산결과로 data2 가 반환된다. 따라서 다음과 같은 형태로 조건 연산자를 구성할 수 있다.

int num3 = (num1>num2) ? (num1) : (num2);

이 경우, 대입 연산자보다 조건 연산자의 우선순위가 높으므로 조건 연산자가 먼저 진행된다. 따라서 num1>num2 가 참이라면 연산의 결과로 num1 이 반환되어 다음의 형태가 된다.

int num3 = num1;

반대로 num1>num2 가 거짓이라면 연산의 결과로 num2 가 반환되어 int num3 = num2;가 된다.

이와 관련해서 아래의 예제를 확인해보자.

```
#include <stdio.h>

int main(void)
{
        int num, abs;
        printf("정 수  입 력 : ");
        scanf("%d", &num);

        abs = num>0 ? num : num*(-1);
        printf("절 댓 값 : %d \n", abs);
        return 0;
}
```

```
정 수  입 력 : -23
절 댓 값 : 23
```

위 예제 9 행에서 보이는 바와 같이 조건의 만족여부에 따라서 반환해야 할 값을 명시할 위치에 num*(-1)과 같은 연산식이 올 수 있으며, 이러한 경우 연산식의 계산결과가 반환이 된다. 즉, 위 예제 9 행의 경우 num>0 이 거짓이라면 num*(-1)연산의 결과가 반환되어서, 변수 abs를 초기화하게 된다.

**5. 반복문의 생략과 탈출: continue & break**

- break! 이제 그만 빠져나가자!

break는 반복문을 탈출할 때 사용하는 키워드로서, 다음의 형태로 문장을 구성한다.

break;

위의 break문이 실행되면, break문을 가장 가까이서 감싸고 있는 반복문 하나를 빠져 나오게 된다. 그럼 다음 예제를 통해서 break문의 기능을 확인해보자. 이는 1+2+3+...+n의 결과가 최초로 5000 을 넘길 때의 n을 구하는 예제이다.

```
#include <stdio.h>

int main(void)
{
        int sum=0, num=0;

        while(1)
        {
                sum+=num;
                if(sum>5000)
                        break; // break문 실 행 , 따 라 서  반 복 문  탈 출
                num++;
        }

        printf("sum: %d \n", sum);
        printf("num" %d \n", num);
        return 0;
}
```

```
sum: 5050
num: 100
```

위 예제에서 주목할 부분은 10, 11 행이다. while문을 돌다가 sum이 5000 보다 커지면 if문의 조건이 참이 되어

break문을 실행하게끔 되어있다. 그런데 여기서 break문이 if문과 함께 쓰였다고 해서 if문을 빠져나오는 것으로 오해하면 안된다. break문은 자신을 감싸는 가장 가까운 위치의 반복문 하나를 빠져나가는데 사용이 된다.

- continue! 나머지 생략하고 반복조건 확인하러 가자!

continue를 이용한 문장의 구성은 다음과 같으며, 이 역시 break문과 마찬가지로 반복문 안에 삽입이 된다.

**continue;**

반복문 안에서 위의 문장을 실행하게 되면, 실행중인 위치에 상관없이 반복문의 조건검사 위치로 이동을 한다. 그리고 검사결과 반복조건이 여전히 참이라면 반복영역을 다시 실행하게 된다.

"실행된 continue문의 이후를 이어서 실행하는 것 아닌가?"

아니다. continue문의 이후는 생략을 하고, 다시 실행하게 된다. 그럼 이와 관련해서 예제를 제시하겠다. 이 예제에서는 1 이상 20 미만의 정수를 출력하되, 2 의 배수와 3 의 배수를 출력에서 제외시키는 프로그램이다.

```
#include <stdio.h>

int main(void)
{
        int num;
        printf("Start! ");

        for(num=1; num<20; num++)
        {
                if(num%2==0 || num%3==0)
                        continue;
                printf("%d", num);
        }
        printf("end! \n");
        return 0;
}
```

```
Start! 15711131719end!
```

실행결과를 통해서도 알 수 있듯이 10 행의 조건이 만족되어서 11 행이 실행되면, 반복문의 나머지에 해당하는 12 행은 실행되지 않고, 다시 조건검사를 위해서 8 행으로 이동한다. 물론 이로 인해서 num의 값이 다시 0 으로 초기화되는 것은 아니다. 예를 들어서 num이 6 인 경우에는 11 행의 continue문을 실행하게 되는데, 이 때 12 행을 건너 뛰고 num의 값이 7 로 증가한 상태에서 반복영역을 처음부터 재실행하게 되는 것이다.

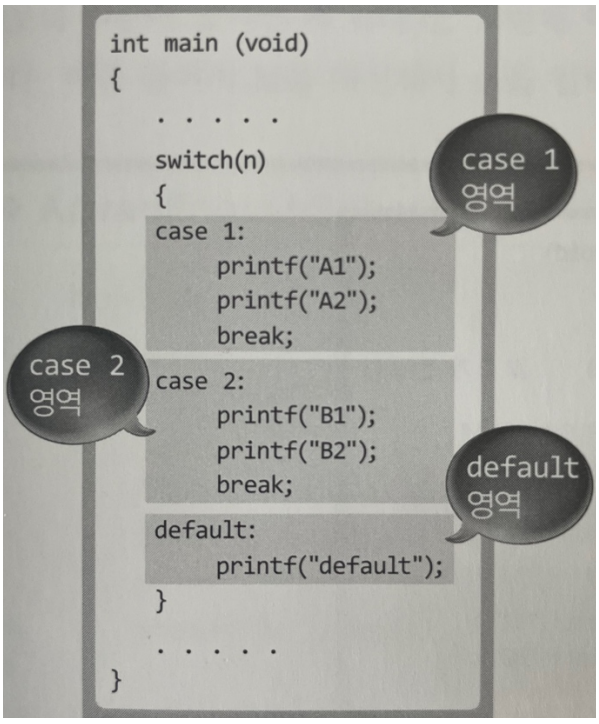
### 6. switch문에 의한 선택적 실행과 goto문

switch문은 if else if else와 유사한 측면이 있다. 때문에 경우에 따라서는 이를 대체하기도 한다. 하지만 사용할 수 있는 영역은 If else if else에 비해 제한적이다.

- switch문의 구성과 기본기능

```
#include <stdio.h>

int main(void)
{
        int num;
        printf("1이상 5이하의 정수 입력 : ");
        scanf("%d", &num);

        switch(num)
        {
        case 1:
                printf("1은 one \n");
                break;
        case 2:
                printf("2는 two \n");
                break;
        case 3:
                printf("3은 three \n");
                break;
        case 4:
                printf("4는 four \n");
                break;
        case 5:
                printf("5는 five \n");
                break;
        default:
                printf("몰라 \n");
        }
        return 0;
}
```

```
1이상 5이하의 정수 입력 : 3
3은 three
```



위 그림에서 보이는 switch(n)에서의 n은 switch문으로 전달되는 인자의 정보이다. 이 n은 정수형 변수이어야 하는데(이 정수형 변수에는 char형도 포함된다), 대표적으로 int형 변수가 위치하게 된다. 그리고 이 n에 저장된 값에 따라서 실행할 영역이 결정된다. n에 저장된 값이 1 이면 case 1 의 영역을 실행하게 되고, n에 저장된 값이 2 이면 case2 의 영역을 실행하게 된다. 참고로 여기서 case1 과 case2 가 의미하는 바는 각각 다음과 같다.

"n을 통해 전달된 정수의 값이 1 인 경우", "n을 통해 전달된 정수의 값이 2 인 경우"

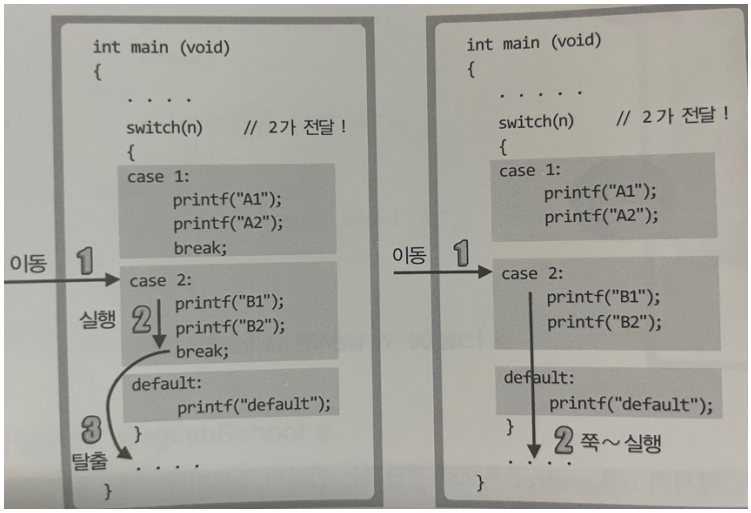
따라서 switch문에 1 이 전달되면 case1 의 영역이 실행되어 문자열 "A1"과 "A2"가 출력되고, switch문에 2 가 전달되면 case2 의 영역이 실행되어 문자열 "B1"과 "B2"가 출력된다. 이렇듯 case문은 위치를 표시하는데 사용되므로 "레이블"이라 한다.

그리고 필요에 따라서 case 레이블은 얼마든지 추가할 수 있으며, 위의 그림과 같이 굳이 case 레이블의 상수 값을 1 부터 시작하지 않아도 된다. 또한 레이블의 끝에는 세미콜론이 아닌 콜론을 붙이도록 되어 있으니, 세미콜론을 붙이지 않도록 주의해야 한다.

이제 그림의 하단부에 위치한 default 레이블을 보자. 이것이 의미하는 바는 If..else if..else의 마지막 else와 유사하다. 즉 일치하는 Case레이블이 없을 경우에 실행되는 영역을 표시할 때 사용된다(불필요하면 생략 가능하다.) 따라서 switch문에 1 과 2 가 아닌 다른 값이 전달되면, default 영역이 실행되어 문자열 default가 출력된다.

그럼 중간에 끼어 있는 break문은 무엇일까?

break문은 반복문을 탈출하는 용도로 사용된다. 그런데 switch문을 탈출하는 용도로도 사용된다. switch문이 영역별로 구분되어 해당 영역만 실행되도록 돕는 것은 break문이다. 그럼 break문의 유무에 따른 실행흐름의 차이는 어떻게 될까? 다음 그림을 통해서 그 차이를 설명하겠다.




위 그림에서는 break문이 있을 때와 없을 때의 실행 흐름을 switch문에 2 가 전달된 상황을 가정하여 설명하고 있다. 위 그림의 오른편에서 보이듯이 switch문에 2 가 전달되면, case2 로 이동하여, 그 이하를 모두 실행해버린다. 따라서 default의 영역에 있는 문장들도 실행이 된다. break문이 없다면 말이다. 이것이 바로 switch문의 기본 동작방식이다.

그럼 이번에는 오른쪽 switch문에 1 이 전달되었다고 가정해보자. 이 경우에는 case1 의 위치로 이동하여 그 이후를 전부 실행하게 된다. 따라서 문자열 A1, A2, B1, B2 그리고 default까지 출력된다. 정리하면, break문이 필요한 이유는 실행영역을 묶기 위함이다.

- break문을 생략한 형태의 switch문 구성

이쯤 되면 break문을 붙여야 하는 불편함에 대해서 불만이 생길 수 있다.

"해당 case문만 실행되도록 switch문의 동작방식을 결정해 놓았다면, Break문을 일일히 붙이는 수고를 덜 수 있었을 텐데"

그러나 이어서 소개하는 예제를 보면 이런 생각이 조금은 바뀔 것 이다. 이 예제는 프로그램 사용자가 M을 입력하면 morning이라는 단어가, A를 입력하면 afternoon이라는 단어가, 그리고 E를 입력하면 evening이라는 단어를 출력해준다. 그런데 이 프로그램은 소문자를 입력해도 해당단어를 출력해준다. 예를들어 m을 입력해도 morning이라는 단어를 출력해준다.

```
#include <stdio.h>

int main(void)
{
        char sel;
        printf("M 오전 , A 오후 , E 저녁  \n");
        printf("입력 : ");
        scanf("%c", &sel);

        switch(sel)
        {
        case 'M':
        case 'm':
                printf("Morning \n");
                break;
        case 'A':
        case 'a':
                printf("Afternoon \n");
                break;
        case 'E':
        case 'e':
                printf("Evening \n");
                break; // 사실 불필요한 break문
        }
        return 0;
}
```

```
M 오전 , A 오후 , E 저녁
입력 : M
Morning
```

12, 13 행의 case레이블을 보면, 이 둘은 사실 다음과 같이 한 줄에 표시할 수도 있다.

case 'M': case 'm':

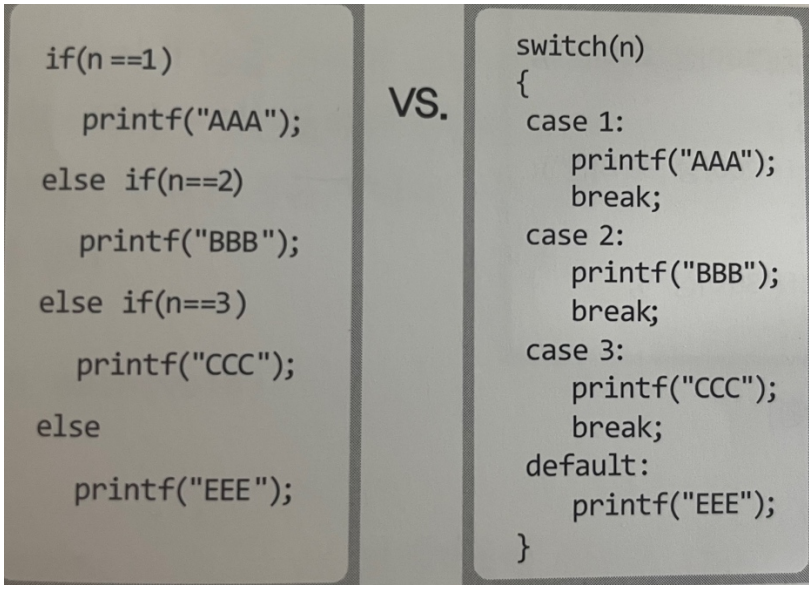
정리하면, switch문에서 break문은 선택적으로 입력을 해야 한다. break문이 삽입되어서 유용한 경우도 있지만, break문이 삽입되지 않아서 유용한 경우도 있기 때문이다.

- switch vs ifelseifelse

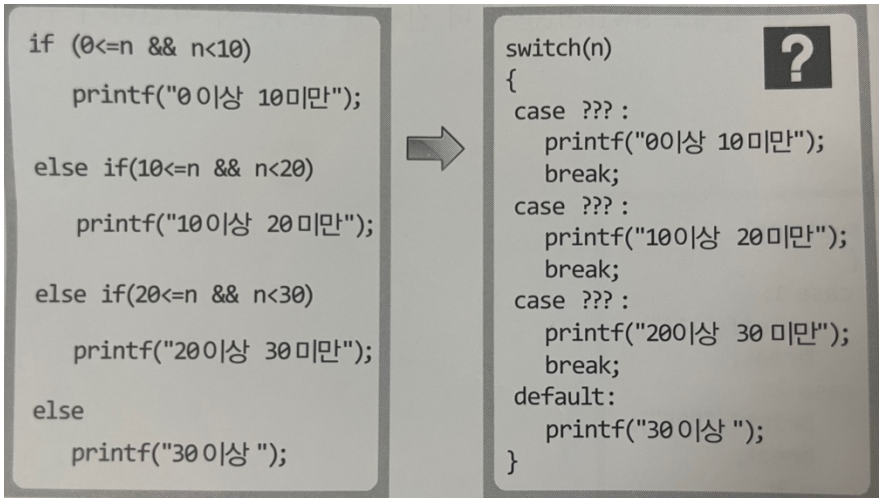
지금까지 switch문에 대해서 공부하였다. 따라서 switch문으로 구현된 모든 예제는 if else if else를 통해서도 구현할 수 있음을 알았을 것이다. 그렇다면 어떤 경우에 switch문을, 그리고 어떤 경우에 ifelseifelse를 사용해야 하는 것일까? 이와 관련해서 첫번째 기준을 제시하겠다.

"분기의 경우 수가 많아지면 가급적 switch문으로 구현을 한다."

switch문의 장점은 if else if else에 비해서 간결해 보인다는 것이다. 따라서 분기의 수가 많아지면 switch문을 사용하는 것이 좋다. 그럼 실제로 switch문이 더 간결해 보이는지 확인하기 위해서 아래의 그림을 제시하겠다.




위 그림의 if else if else문과 switch문이 하는 일은 같다. 하지만 switch문이 더 간결해 보인다.
선택 두 번째 기준을 제시하겠다.
"switch문으로 구현할 수 있는 조건의 구성에는 한계가 있다. 따라서 switch문으로 표현하기 애매한 상황에서는 if else if else를 사용해야 한다."
다음 그림에서 보이는 경우가 switch문으로 표현하기 애매한 상황이다.




위와 같이 if else if else는 참과 거짓의 연산결과를 이용해서 실행할 영역을 결정하는 방식이기 때문에 이 모든 것은 switch문으로 대신하는 데는 한계가 있다. 따라서 분기의 수가 많다고 해서 switch문만 고집할 필요는 없다.

- goto

goto는 그 이름이 의미하듯이 프로그램의 흐름을 원하는 위치로 이동시킬 때 사용하는 키워드이다. 아주 오래전에는 goto의 필요성에 대한 논쟁도 있었지만, 이제는 goto의 사용에 부정적으로 결론이 난듯한 분위기이다. 실제로 근래에 출간되는 c언어 서적 중에는 goto를 아예 언급조차 하지 않는 서적도 있을 정도이다. 그렇다면 goto의 사용을 부정적으로 보는 이유는 무엇일까? goto문의 가장 큰 단점은 프로그램의 자연스러운 흐름을 방해한다는 것이다. c언어와 같은 절차지향 프로그래밍 언어에서 프로그램의 흐름을 방해하거나 복잡하게 하는 것은 아주 큰 단점이 된다. 그만큼 단순한 흐름이 중요하기 때문이다. 그런데다가 goto문을 써야만 해결할 수 있는 문제의 상황도 딱히 존재하지 않는다. 물론 goto문을 써서 개선시킬 수 있는 상황이 아예 없는 것은 아니지만, 극히 제한적이며 또 설득력이 강한 것도 아니다. 그래서 사람들은 이러한 goto문의 사용을 가급적 자제하거나 아예 사용하지 말자고 결론을 내리게 되었다. 그래서 나는 간단하게 goto문을 공부해보자 한다.

goto문의 기본구성은 다음과 같다.

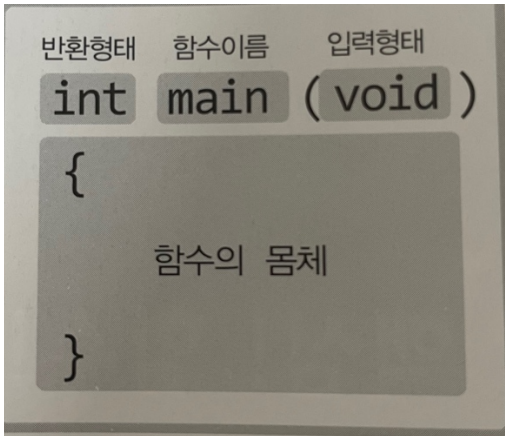
```
int main(void)
{
        ....
rabbit: // 위치 지정에 사용된 rabbit이라는 이름의 레이블
        ....
        goto rabbit: // 레이블 rabbit의 위치로 이동
        ....
}
```

위의 코드에서 보이듯이 이동의 위치는 레이블로 표시한다. 그리고 해당 레이블을 대상으로 goto문을 구성하게 된다. 그럼 완성된 예제를 통해서 실제로 실행의 위치가 이동을 하는지 확인해보겠다.

```
#include <stdio.h>

int main(void)
{
        int num;
        printf("자연수 입력 : ")
        scanf("%d", &num);

        if(num==1)
                goto ONE;
        else if(num==2)
                goto TWO;
        else
                goto OTHER;

        ONE:
                printf("1을 입력하셨습니다! \n");
                goto END;
        TWO:
                printf("2를 입력하셨습니다! \n");
                goto END;
        OTHER:
                printf("3 혹은 다른 값을 입력하셨습니다! \n");

        END:
                return 0;
}
자연수 입력 : 3
3 혹은 다른 값을 입력하셨습니다!
```

위의 예제 16,19,22,25 행에서는 레이블이 존재한다. 그런데 레이블은 실행의 대상이 아닌, 위치를 표시하는 역할로 사용되는 표식이기 때문에 대부분의 편집기상에서 왼쪽 정렬을 시킨다. 그래서 프로그래머들이 쉽게 구분할 수 있도록 돕는다.

### 8. 함수를 정의하고 선언하기




- 함수를 만드는 이유

해결해야 할 문제의 규모가 크거나 그 형태가 복잡한 경우 사람들이 흔히 하는 말이 있다.

"자, 천천히 하나씩 해결해 나가자"

급하다고 해서 한꺼번에 모든 문제를 해결하려 들지 말고, 작은 문제부터 하나씩 해결해 나아가자는 뜻이다. 그런데 이러한 방법은 실제로 효과를 발휘한다. 크고 복잡한 문제를 작게 나눠서 하나씩 해결해 나가는 것이 보다 빠르게, 그리고 정확히 문제를 해결하는 원칙이 된다.

프로그램의 구현은 복잡한 문제를 해결하는 것에 비유할 수 있다. 따라서 main이라는 하나의 함수 안에서 문제를 해결하려 드는 것은 아무런 대책 없이 무작정 문제를 해결하려 드는 것과 같다. 프로그램을 구현하려면, 구현에 필요한 기능들을 분석하고 그 분석결과를 바탕으로 작은 크기의 함수들을 디자인해야 한다. 그리고 이들을 이용해서 하나의 프로그램을 엮어가야 한다. 실제로 작은 크기의 함수를 구현해야 하는 것은 복잡하고 커다란 문제를 작게 쪼개서 해결하는 것에 비유할 수 있다. 그리고 프로그램을 작은 크기의 함수들로 나눠서 구현하게 되면, 문제의 발생 및 프로그램의 요구사항 변경으로 인한 소스코드의 변경이 필요한 경우에, 변경의 범위를 축소 및 제한할 수 있다. 이러한 장점은 여러분이 앞으로 프로그램을 실제 개발하면서 느끼게 될 것이다.

- 함수의 입력과 출력: printf 함수도 반환을 한다

C언어의 함수는 전달인자가 없거나 반환 값이 없는 경우도 있다. 물론 둘 다 없는 경우도 존재한다. printf 함수를 예로 들어보면, 이 함수는 전달되는 인자정보들을 참조하여 데이터를 모니터에 출력하는 기능을 지닌다. 따라서 굳이 값을 반환할 필요가 없어 보인다. 실제로 값을 반환하지 않는다고 해서 문제 될 것도 없다. 그래도 여전히 함수라 할 수 있으며 모니터에 뿌려지는 데이터들을, 값의 반환을 대신하는 출력의 일종으로 볼 수 있다. 이렇듯 불필요하다면 입력이나 출력이 존재하지 않는 형태의 함수를 정의하는 것도 가능하다.

"입력 또는 출력이 없는 함수도 존재한다(만들 수 있다)"

printf 함수는 실제로 값을 반환하지 않을까? 관심 밖의 일이라서 모르고 있었지만 printf 함수도 실제로는 값을 반환한다. 그럼 다음 예제를 통해서 이를 확인해보겠다.

```
#include <stdio.h>

int main(void)
{
        int num1, num2;
        num1=printf("12345\n");
        num2=printf("I love my home\n");
        printf("%d %d \n", num1, num2);
        return 0;
}
```

```
12345
I love my home
6 15
```

실행 결과에서 보이듯이 Printf함수는 ₩n 문자를 포함하여 모니터에 출력한 문자열의 길이를 반환한다. 이제

어느 정도의 함수의 특징을 이해했을 것이다.

- 전달인자의 유무와 반환 값의 유무에 따라서 함수를 네 개의 형태로 나누자.

함수의 이름을 정하고, 기능을 구현하고, 전달인자와 반환 값을 결정짓는 것이 함수를 정의하는 것이다. 여기서는 다양한 형태의 함수를 정의해 봄으로써 우리들 스스로가 필요한 함수를 정의할 수 있는 능력을 키워보고자 한다. 따라서 함수의 유형을 다음과 같이 총 4 가지로 나누겠다.

- 유형 1 : 전달인자 있고, 반환 값 있다! 전달인자(O), 반환 값(O)
- 유형 2 : 전달인자 있고, 반환 값 없다! 전달인자(O), 반환 값(X)
- 유형 3: 전달인자 없고, 반환 값 있다! 전달인자(X), 반환 값(O)
- 유형 4: 전달인자 없고, 반환 값 없다! 전달인자(X), 반환 값(X)

그리고 각 유형별 함수를 하나씩 정의해 보겠다.

- 다양한 형태의 함수 정의 1: 전달인자와 반환 값이 모두 있는 경우

전달인자와 반환 값이 모두 있는 경우, 이는 가장 일반적인 형태의 함수이다. 여기서는 덧셈 기능을 지니는 함수를 정의해보겠다. 일단 전달인자는 두 개가 되어야 한다. 덧셈이라는 것이 둘 이상의 수를 더하는 것 아닌가? 그리고 반환 값도 존재해야 한다. 덧셈을 했다면 그에 따른 결과를 반환해야 할 것 아닌가? 그래서 구현할 함수의 특징을 다음과 같이 정리하였다.

- 전달인자는 int형 정수 둘이며, 이 둘을 이용한 덧셈을 진행한다
- 덧셈결과는 반환이 되며, 따라서 반환형도 int형으로 선언한다.
- 마지막으로 함수의 이름은 Add라 하자

위의 조건을 만족시켜서 함수를 정의하면 다음과 같다.

```
int Add(int num1, int num2)
{
        int result=num1+num2;
        return result; // result에 저장된 값을 Add 함수를 호출한 영역으로 전달
}
```

(int num1, int num2)는 매개변수의 선언이다. 이는 함수호출 시 전달되는 인자를 저장할 변수의 선언이다.

이어서 몸체부분을 보면, 함수 몸체의 첫 번째 행에서 하는 일은 num1 과 num2 의 값을 더해서 그 결과를 변수 result에 저장하는 일이다. 그리고 result에 저장된 값을 **return result;** 로 표시되어 있는 영역에서 반환한다. 그런데 그 반환되는 값의 자료형이 int형이므로 맨 윗행에서 반환형이 int형으로 선언되었다. 마지막으로 Add에는 지금까지 설명한 함수의 이름이 명시 되어있다.

참고로 함수호출 시 전달할 수 있는 인자의 수는 여러 개가 될 수 있지만 반환할 수 있는 값의 수는 하나이다. 따라서 반환의 대상을 결정할 때 더 신경을 써야 한다. 그럼 이로써 Add함수에 대한 설명을 마치고, 이번에는 Add함수를 실제로 호출하는 예제를 보이겠다. 단, 아래의 예제에서는 Add함수를 더 간략히 정의하였다.

```
#include <stdio.h>

int Add(int num1, int num2)
{
        return num1+num2;
}

int main(void)
{
        int result;
        result = Add(3, 4);
        printf("덧셈 결과 1: %d \n", result);
        result = Add(5, 8);
        printf("덧셈 결과 2: %d \n", result);
        return 0;
}
```

```
덧셈 결과 1: 7
덧셈 결과 2: 13
```

위 예제의 5 행은 다음과 같다.

**return num1+num2;**

이렇듯 매개변수를 대상으로 직접 덧셈연산을 하여 그 결과를 바로 반환할 수도 있다. 이러한 경우 num1 과 num2 를 대상으로 먼저 덧셈연산이 이뤄지고, 그 때 얻어지는 결과 값을 반환하게 된다. 이어서 Add 함수를 호출하는 11 행을 보자.

**result = Add(3, 4);**

값이 변환된다는 것은 함수의 호출문이 반환 값으로 대체되는 것으로 이해할 수 있다. 즉, 위의 문장에서 Add 함수가 종료되면서 7 을 반환하면 다음과 같이 대입연산만 남은 형태가 된다.

**result = 7;**

그래서 변수 result에는 7 이 저장되는 것이다.

- 다양한 형태의 함수 정의 2: 전달인자나 반환 값이 존재하지 않는 경우

이어서 위의 예제에 몇몇 함수를 추가해서 예제의 완성도를 높여보겠다. 먼저 그 첫 번째 순서로 printf를 대신할 수 있는 함수를 정의하고자 한다. printf 함수는 서식을 지정해야 하기 때문에 상대적으로 빈번히 호출하기가 번거로운 함수이다. 그래서 서식지정을 매번 하지 않아도 되도록 다음과 같이 함수를 정의하였다.

```
void ShowAddResult(int num) // 인자전달 (O), 반환 값(X)
{
        printf("덧셈결과 출력 : %d ₩n", num);
}
```

위의 함수 정의에서 반환형이 void로 선언되었다. 여기서 사용된 void에는 반환하지 않는다 라는 뜻이 담겨있다. 실제로 함수의 몸체부분을 보면 return문이 없음을 알 수 있다. 이어서 프로그램 사용자로부터 정수를 입력 받을 때 호출하는 함수를 정의하고자 한다. 이 함수는 scanf 함수보다 호출하기 편리하도록 다음과 같이 정의하였다.

```
int ReadNum(void) // 인자전달(X), 반환 값(O)
{
        int num;
        scanf("%d", &num);
        return num;
}
```

위의 함수에서는 매개변수의 선언 위치에 void 선언이 등장하였다. 여기서 사용된 void는 인자를 전달하지 않는다 라는 뜻이 담겨있다. 따라서 함수를 호출할 때 실제로 인자를 전달하면 안된다. 마지막으로 프로그램의 사용방법을 소개하는 함수를 다음과 같이 정의하고자 한다. 이 함수는 단순히 메시지를 전달하는 함수이기 때문에 인자의 전달도 값의 반환도 불필요하다. 따라서 매개변수도, 그리고 반환형도 void로 선언되었다.

```
void HowToUseThisProg(void) // 인자전달(X), 반환 값(X)
{
        printf("두 개의 정수를 입력하시면 덧셈결과가 출력됩니다. ₩n");
        printf("자 그럼 두 개의 정수를 입력하세요. ₩n");
}
```

이렇게 해서 반환형의 유무와 매개변수의 유무에 따라서 구분할 수 있는 함수의 유형을 모두 살펴보았다. 그럼 이제 지금까지 정의한 함수들을 모두 포함하여 위 예제를 확장하겠다.

```
#include <stdio.h>

int Add(int num1, int num2) // 인자전달 (O), 반환 값 (O)
{
        return num1+num2;
}

void ShowAddResult(int num) // 인자전달 (O), 반환 값(X)
{
        printf("덧셈결과 출력 : %d \n", num);
}

int ReadNum(void) // 인자전달 (X), 반환 값 (O)
{
        int num;
        scanf("%d", &num);
        return num;
}

void HowToUseThisProg(void) // 인자전달 (X), 반환 값(X)
{
        printf("두 개의 정수를 입력하시면 덧셈결과가 출력됩니다. \n");
        printf("자, 그럼 두 개의 정수를 입력하세요. \n");
}

int main(void)
{
        int result, num1, num2;
        HowToUseThisProg();
        num1=ReadNum();
        num2=ReadNum();
        result = Add(num1, num2)
        ShowAddResult(result);
        return 0;
}
두 개의 정수를 입력하시면 덧셈결과가 출력됩니다.
자, 그럼 두 개의 정수를 입력하세요.
12
34
덧셈결과 출력 : 46
```

- return이 지니는 두 가지 의미 중 한가지 의미만 살리기

키워드 return은 값을 반환하면서 함수를 빠져나갈 때 사용된다. 즉, return문에는 다음 두 가지 의미가 담겨있다.

- 함수를 빠져나간다.
- 값을 반환한다.

간혹 반환형이 void로 선언된 함수에서는 return문을 사용할 수 없는 것으로 아는 경우가 있는데, 반환형이 void인 함수에서도 다음의 형태로 return문을 삽입할 수 있다.

```
void NoReturnType(int num)
{
        if(num<0)
                return; // 값을 반환하지 않는 return문
        ....
}
```

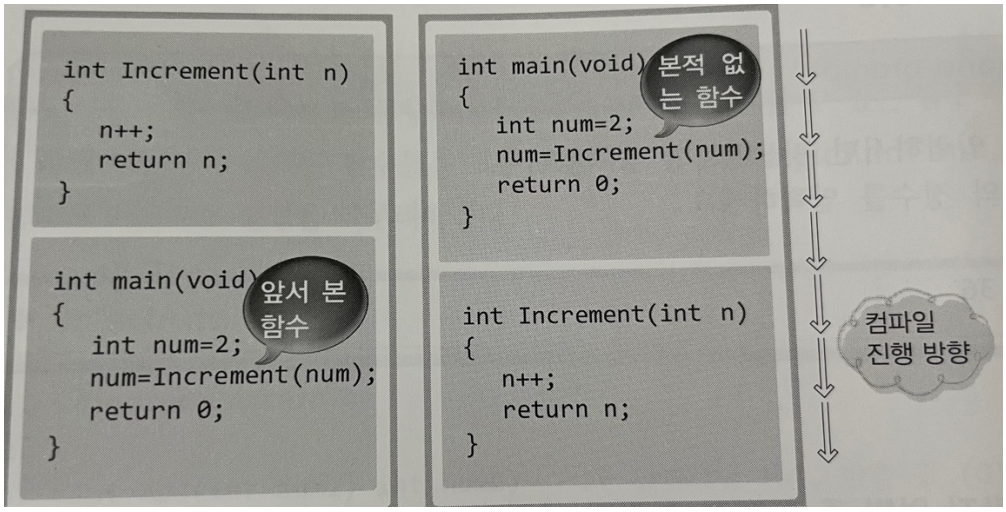
위의 return문에는 반환할 값이 명시되어 있지 않는데, 이는 return의 두 가지 의미 중 다음 한가지 의미만 담아서 선언했기 때문이다.

"값의 반환 없이 그냥 함수를 빠져나간다"

이렇듯 값을 반환하지 않고 함수를 빠져나가는 목적으로만 선언이 되었으므로, 반환형이 void인 함수에도 삽입이 가능하다. 그리고 실제로 반환형이 void인 함수에 이러한 return문이 삽입되는 경우도 간혹있다.

- 함수의 정의와 그에 따른 원형의 선언

함수의 위치를 결정할 때에도 주의를 기울여야 하는데, 이와 관련해서 다음 그림을 보면서 설명을 하겠다.




위 그림은 두 가지 형태의 프로그램 구성을 보이는데, 왼쪽의 경우는 Increment 함수의 정의가 Increment 함수의 호출문 보다 먼저 등장하고 있다. 따라서 컴파일러는 main 함수에 존재하는 Increment 함수의 호출문을 보면서 다음과 같이 판단하고 무리 없이 컴파일 한다.

"앞서 내가 컴파일 했던 Increment 함수를 호출하는구나"

하지만 오른쪽의 경우에는 문제가 발생한다. Increment 함수의 정의에 앞서 Increment 함수의 호출문이 먼저 등장했기 때문이다. 위에서 아래로 컴파일을 진행하는 컴파일러는 Increment 함수의 호출문을 보면서 컴파일 에러를 일으킨다. 비록 뒤에 이어서 increment 함수의 정의가 등장하지만 C 컴파일러는 이를 문제시 삼는다. 즉 함수는 호출되기 전에 미리 정의되어야 한다. 하지만 그림의 오른편에 있는 구성으로 프로그래밍을 할 수 있는 방법이 있다. 그것은 컴파일러에게 increment 함수가 뒤에서 나온다고 알려주는 것이다.

그리고 이러한 내용을 담은 선언에 해당하는 것이 다음 문장이다. 이 문장에는 함수의 이름과 반환형 그리고 매개변수 정보가 모두 포함되어 있음에 주목하기 바란다.

```
int Increment(int n);
```

따라서 다음 아래와 같이 예제를 작성할 수도 있다.

```
int Increment(int n);
int main(void)
{
        int num=2;
        num=Increment(num);
        return 0;
}
int Increment(int n)
{
        n++
        return n;
}
```

이로써 main 함수에서 호출하는 Increment 함수의 정의가 main 함수의 뒤에 올 수 있게 되었다. 참고로 함수의 선언에는 매개변수의 갯수 및 자료형 정보만 포함되면 되기때문에 다음과 같이 매개변수의 이름을 생략해서 선언하는 것이 가능하다.

```
int Increment(int n);
```

마찬가지로 이전의 예제에서 정의한 Add함수의 선언은 다음 두 가지 모두가 될 수 있다.

int Add(int num1, int num2); // 매개변수의 이름을 포함한 선언

int Add(int, int) // 매개변수의 이름을 생략한 표현

- 다양한 종류의 함수 정의하기

이번에 소개하는 예제는 다음의 상황을 소개하기 위한 것이다.

"하나의 함수 내에 둘 이상의 return문이 존재하는 경우"

그러나 return문이 실행되면 값을 반환하면서 함수를 빠져나간다는 기본원칙을 통해서 쉽게 이해할 수 있는 예제이다.

```
#include <stdio.h>
int number(int num1, int num2);

int main(void)
{
        printf("3과 4중에서 큰 수는 %d이다. \n, number(3, 4));
        printf("7과 2중에서 큰 수는 %d이다. \n, number(7, 2));
        return 0;
}

int number(int num1, int num2)
{
        if(num1>num2)
                return num1;
        else
                return num2;
}
```

```
3과 4중에서 큰 수는 4이다.
7과 2중에서 큰 수는 7이다.
```

위 예제에 정의된 함수 number는 if의 조건 num1>num2 를 만족하면 다음 문장을 함수의 중간에서 실행한다.

return num1;

이렇듯 함수의 중간에도 얼마든지 return문이 올 수 있다. 그럼 이번에는 6행과 7행을 관찰하자. 이 두가지 printf 함수 호출문의 두 번째 인자로는 %d에 대응하는 정수가 와야 하는데, 이 위치에 다음과 같이 '함수의 호출'이 대신 자리를 잡고 있다. 이 경우에는 함수의 반환 값이 전달인자를 대신하게 된다. 즉, number 함수가 먼저 호출되어 다음의 형태로 printf 함수의 호출이 진행된다.

printf("3 과 4 중에서 큰 수는 %d이다. ₩n", 4);

printf("7 과 2 중에서 큰 수는 %d이다. ₩n", 7);

참고로 위 예제에서 보인 함수 number는 완전하지 못하다. 동일한 값의 두 정수가 전달되었을 때, 값이 동일함을 알리지 못하는 구조로 정의되어 있다. 따라서 이후에 약간의 수정이 불가피하다.

마지막으로 프로그램 사용자가 입력한 두 개의 정수 중에서 절댓값이 큰 수를 반환하는 함수를 정의해보겠다. 그런데 이 함수는 인자로 전달된 정수의 절댓값을 계산하는 과정을 거쳐야하는데, 이를 대신할 수 있는 함수를 별도로 정의하고, 중간에 이를 호출하는 형태로 정의해보겠다.

```
#include <stdio.h>
int com(int num1, int num2); // 절댓값이 큰 정수 변환
int val(int num); // 전달인자의 절댓값을 반환

int main(void)
{
        int num1, num2;
        printf("두 개의 정수 입력 : ");
        scanf("%d %d", &num1, &num2);
        printf("%d와 %d중 절댓값이 큰 정수 : %d \n,
                num1, num2, com(num1, num2));
        return 0;
}

int com(int num1, int num2)
{
        if(val(num1) > val(num2))
                return num1;
        else
                return num2;
}

int val(int num)
{
        if(num<0)
                return num * (-1);
        else
                return num;
}
```

```
두 개의 정수 입력 : -5
2
-5와 2중 절댓값이 큰 정수 : -5
```

먼저 위 예제의 17~20 행의 if~else문을 보자. 이 if~else문의 특징은 조건의 비교에 함수의 호출이 존재한다는 것이다. 다음의 if~else문은,

if(val(num1) > val(num2))

두 번의 val 함수호출 후에 반환되는 정수를 대상으로 다음과 같이 비교연산을 진행하게 된다.

if( 5 > 2)

그리고 위 예제의 특징은 호출된 함수 내에서 또 다른 함수를 호출한다는 점이다. Main 함수에서 절댓값 비교를 위해서 함수 com을 호출했는데, 이 함수의 몸체 부분에서 절댓값을 얻기 위해서 함수 val을 다시 호출하였다. 이렇듯 main 함수가 아닌 다른 함수에서도 필요에 따라서 얼마든지 함수를 호출할 수 있다. 즉, main을 포함한 모든 함수는 조건 및 상황에 관계없이 다른 함수를 호출할 수 있다.

**9. 변수의 존재기간과 접근범위 1: 지역변수**

변수의 선언위치와 함수는 깊은 관계가 있다. 변수는 선언되는 위치에 따라서 크게 전역변수와 지역변수로 나뉜다. 그리고 이 둘은 다음 두 가지에 대해서 차이점을 보인다.

- 메모리상에 존재하는 기간
- 변수에 접근할 수 있는 범위

- 함수 내에만 존재 및 접근 가능한 지역변수

지역변수에서 말하는 지역이란 중괄호에 의해 형성되는 영역을 뜻한다. 따라서 **중괄호 내에 선언되는 변수는 모두 지역변수**다. 그런데 이러한 지역변수는 선언된 지역내에서만 유효하다는 특성을 지닌다. 그럼 이와 관련된 내용을 아래의 코드를 통해서 설명하겠다.

```c
#include <stdio.h>

int sim(void)
{
        int num=10; // 이후부터 sim의 num dbgy
        num++;
        printf("sim num: %d \n", num);
        return 0; // sim의 numdl 유효한 마지막 문장
}

int sim2(void)
{
        int num1=20; // 이후부터 num1 유효
        int num2=30; // 이후부터 num2 유효
        num1++, num2--;
        printf("num1 & num2: %d %d \n", num1, num2);
        return 0; // num1, num2 유효한 마지막 문장
}

int main(void)
{
        int num=17; // 이후부터 main의 num dbgy
        sim();
        sim2();
        printf("main num: %d \n", num);
        return 0; // main의 num이 유효한 마지막 문장
}
```

```
sim num: 11
num1 & num2: 21 29
main num: 17
```

위 예제의 다음 함수 sum을 먼저 관찰하자. 위의 함수에 선언된 변수 num은 sim이라는 함수의 중괄호 안에 선언되었으므로 지역변수이다. 따라서 이 변수는 선언된 이후부터 sim함수를 빠져나가기 직전까지만 유효하다. 왜냐하면 지역변수는 해당지역을 벗어나면 자동으로 소멸되기 때문이다.

그럼 sim 함수가 호출될 때마다 변수 num은 새롭게 메모리 공간에 할당되나? 그렇다. sim 함수가 10 번 호출되면 총 10 회에 걸쳐서 변수 num은 할당되고 또 소멸된다. 그리고 지역변수는 선언된 지역 내에서만 유효하기 때문에 선언된 지역이 다르면 이름이 같아도 문제가 되지 않는다. 때문에 위 예제의 main 함수 내에도, 그리고 Sim 함수 내에도 동일한 변수 Num이 선언될 수 있는 것이다.

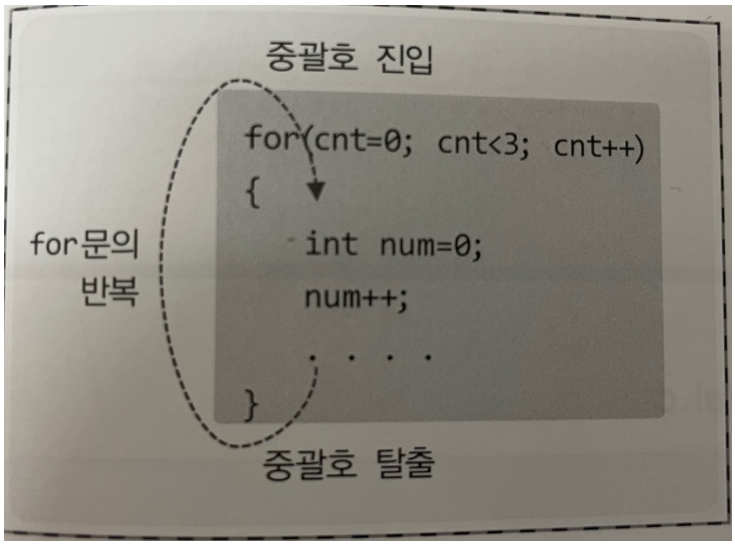
- 다양한 형태의 지역변수

앞서 중괄호 내에 선언되는 변수는 모두 지역변수라고 했다. 그런데 중괄호는 함수의 정의에만 존재하는 것이 아니다. 반복문이나 조건문에도 중괄호는 존재할 수 있다. 즉, 지역변수는 반복문이나 조건문에도 선언이 가능하다. 그럼 이와 관련해서 다음 예제를 보자.

```c
#include <stdio.h>

int main(void)
{
        int cnt;
        for(cnt=0; cnt<3; cnt++)
        {
                int num=0;
                num++;
                printf("%d번째 반복, 지역변수 num은 %d. \n", cnt+1, num);
        }

        if(cnt==3)
        {
                int num=7;
                num++;
                printf("if문 내에 존재하는 지역변수 num은 %d. \n", num);
        }
        return 0;
}
```

```
1번째 반복, 지역변수 num은 1.
2번째 반복, 지역변수 num은 1.
3번째 반복, 지역변수 num은 1.
if문 내에 존재하는 지역변수 num은 8.
```

위 예제에서 제일먼저 선언된 지역변수는 5 행의 cnt이다. 이 변수는 main 함수의 시작부분에서 선언되었다. 따라서 Main함수 내에서는 어디서는 접근 가능하다. 6 행의 for문 내에서도, 13 행의 if문 내에서도 접근 가능하다. 10 행의 출력결과를 통해서 이러한 사실을 확인할 수 있다. 이어서 8 행에서 선언된 지역변수 num을 보자. 이 변수는 for문의 중괄호 내에 선언되었기 때문에 for문내에서 유효한 지역변수가 된다. 그런데 여기 한가지 주의할 것이 있다. 이와 관련해서 다음 그림을 보자.




위 그림은 for문에 의한 반복이 중괄호 내에서 이뤄지는 것이 아니라, 중괄호의 진입과 탈출을 반복하면서 이뤄지는 것임을 말하고 있다. 따라서 반복이 이뤄질 때마다 변수 Num은 메모리상에 할당되고 소멸된다. 그래서 10 행의 출력결과는 항상 1 이다.

마지막으로 3 번째 지역변수는 15 행에서 선언된다. 비록 if문 내에 선언되었지만 이 역시 중괄호 내에 선언되었으므로 if문 내에서만 유효한 지역변수가 된다. 따라서 if문을 벗어나면 소멸되는 지역변수이다.

그리고 지역변수는 외부에 선언된 동일한 이름의 변수를 가리게 된다.

- 지역변수의 일종인 매개변수

함수를 정의할 때 선언하는 매개변수도 지역변수의 일종이다. 따라서 매개변수 역시 지역변수의 특성을 지닌다. 따라서 매개변수는 지역변수이다 라고 말해도 틀리지 않는다. 하지만 그 역은 성립하지 않는다. 왜냐하면 모든 지역변수가 매개변수는 아니기 때문이다.

**9. 변수의 존재기간과 접근범위 2: 전역변수, static 변수, register 변수**

전역변수는 프로그램이 처음 실행되면 메모리 공간에 할당되어서 프로그램이 종료될 때까지 메모리 공간에 남아있는 변수이다.

- 전역변수의 이해와 선언방법

전역변수는 그 이름이 의미하듯이 어디서든 접근이 가능한 변수로써 지역변수와 달리 중괄호 내에 선언되지 않는다. 그럼 다음 예제를 통해서 전역변수의 선언방법과 특징을 살펴보자.

```
#include <stdio.h>
void Add(int val);
int num; // 전역변수는 기본 0으로 초기화됨

int main(void)
{
        printf("num: %d \n", num);
        Add(3);
        printf("num: %d \n", num);
        num++; // 전역변수 num의 값 1 증가
        printf("num: %d \n", num);
        return 0;
}

void Add(int val)
{
        num += val; // 전역변수 num의 값 val만큼 증가
}
```

```
num: 0
num: 3
num: 4
```

위 예제의 3 행에 선언된 변수는 어떠한 중괄호에도 포함되지 않는다. 이렇게 선언되는 변수를 전역변수라 하는데, 전역변수는 다음의 특징을 지닌다.

- 프로그램의 시작과 동시에 메모리 공간에 할당되어 종료 시까지 존재한다.
- 별도의 값으로 초기화되지 않으면 0 으로 초기화된다.
- 프로그램 전체 영역 어디서든 접근이 가능하다.

- 전역변수와 동일한 이름의 지역변수가 선언되면?

전역변수와 동일한 이름의 지역변수가 선언된다면?

"해당 지역 내에서는 전역변수가 가리워지고, 지역변수로의 접근이 이뤄진다.:

```
#include <stdio.h>
int num=1;

int Add(int val)
{
        int num=9;
        num += val;
        return num;
}

int main(void)
{
        int num = 5;
        printf("num: %d \n", Add(3));
        printf("num: %d \n", num+9);
        return 0;
}
```

```
num: 12
num: 14
```

전역변수와 지역변수의 이름은 달리하는 것이 좋다.

- 전역변수, 괜찮은 것 같은데 많이 써도 될까?

전역변수는 프로그램의 구조를 복잡하게 만드는 주범이기 때문에 전역변수의 선언은 가급적 제한해야 한다.

- 지역변수에 static 선언을 추가해서 만드는 static 변수

전역변수와 지역변수 모두에 static 선언을 추가할 수 있다. 지역변수에 static 선언이 붙게 되면, 이는 전역변수의 성격을 지니는 변수가 된다.

- 선언된 함수 내에서만 접근이 가능하다
- 딱 1 회 초기화되고 프로그램 종료 시까지 메모리 공간에 존재한다.

```
#include <stdio.h>

void sim(void)
{
        static int num1=0; //초기화하지 않으면 0 초기화
        int num2 = 0; // 초기화하지 않으면 쓰레기 값 초기화
        num1++, num2++;
        printf("static: %d, local: %d \n", num1, num2);
}

int main(void)
{
        int i;
        for(i=0; i<3; i++)
                sim();
        return 0;
}
```

```
static: 1, local: 1
static: 2, local: 1
static: 3, local: 1
```

위 예제의 sim 함수는 다음과 같이 정의되어 있다.

```
void sim(void)
{
        static int num1=0;
        ....
}
```

static으로 선언된 지역변수는 전역변수와 동일한 시기에 할당하고 소멸된다. 단, 지역변수와 마찬가지로 선언된 함수 내에서만 접근이 가능하다.

- static 지역변수는 좀 써도 될까?

static 지역변수로만 해결이 가능한 문제는 존재하지 않는다. static 지역변수는 매우 쉽게 전역변수로 대체가 가능하기 때문이다. 하지만 static 지역변수는 전역변수보다 좋다. 전역변수와 마찬가지로 프로그램이 종료될 때까지 메모리 공간에 남아있지만, 접근할 수 있는 범위를 하나의 함수로 제한했기 때문이다. 따라서 static 지역변수를 전역변수로 대체하는 일은 없어야 한다. 반대로 전역변수를 static 지역변수로 대체할 수 있다면 대체해서 프로그램의 안전성을 높여야 한다.

- 보다 빠르게, register 변수

지역변수에는 다음과 같이 register라는 선언을 추가할 수 있다. 그리고 이렇게 해서 선언된 변수를 가리켜 레지스터 변수라 한다.

```
int simple(void)
{
        register int num =3;
        ....
}
```

위와 같이 선언이 되면 변수 num은 cpu내에 존재하는 레지스터라는 메모리 공간에 저장될 확률이 높아진다. 왜냐하면 이는 다음과 같은 힌트를 컴파일러에게 전달하는 선언이기 때문이다.

"이 변수는 내가 빈번히 사용하니, 접근이 가장 빠른 레지스터에 저장하는 것이 성능향상에 도움이 될 거야"
레지스터는 cpu내에 존재하는 그 크기가 매우 작은 메모리이다. 하지만 CPU내에 존재하기 때문에 이 메모리에 저장된 데이터를 대상으로 하는 연산은 매우 빠르다. 바로 이러한 레지스터의 활용과 관련해서 컴파일러에게 힌트를 주는 선언이 바로 register 선언이다. 따라서 컴파일러는 이 선언을 힌트로 하여 레지스터의 활용여부를 결정한다. 다시 한번 말하지만 최종결정은 컴파일러가 내린다. 우리가 아무리 레지스터 선언을 추가해도 컴파일러가 합당하지 않다고 판단하면 레지스터에 할당되지 않는다. 반대로 아무런 선언을 하지 않아도 컴파일러가 레지스터에 할당해야겠다고 판단하면 그 변수는 레지스터에 할당된다. 그리고 전역변수에는 레지스터 선언을 추가할 수 없다.

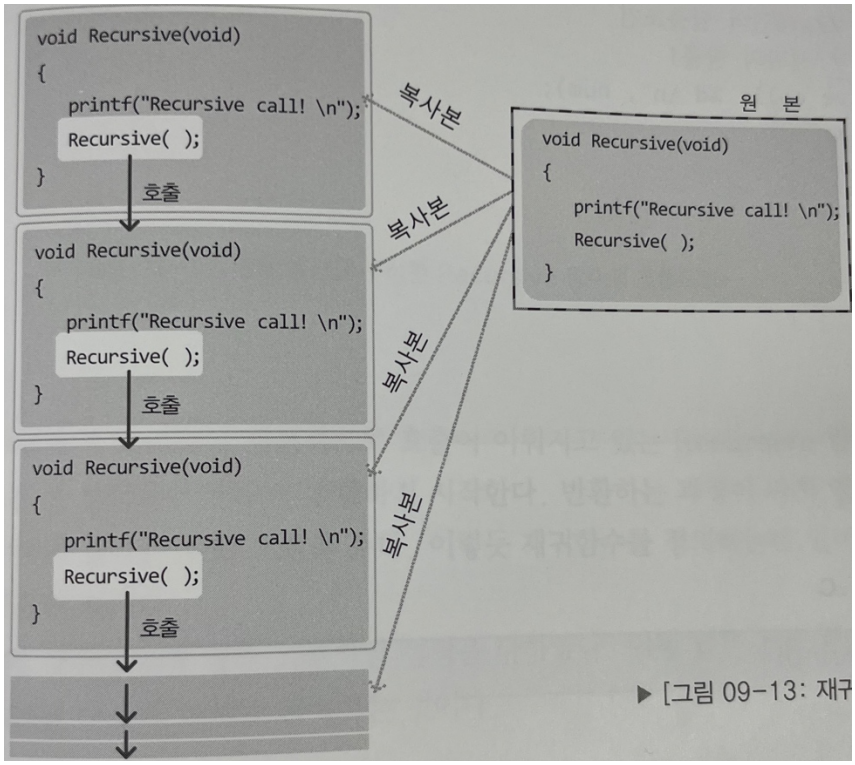
**10. 재귀함수에 대한 이해**

- 재귀함수의 기본적인 이해

재귀함수란 다음과 같이 함수 내에서 자기 자신을 다시 호출하는 함수를 의미한다.

```
void Recursive(void)
{
        printf("Recursive call! ₩n");
        Recursive(); // 나 자신을 재 호출한다.
}
```

그렇다면 위 형태의 함수호출은 어떻게 이해해야할까?




Recursive 함수를 실행하는 중간에 다시 Recursive 함수가 호출되면, Recursive 함수의 복사본을 하나 더 만들어서 복사본을 실행하게 된다.

위 문장의 내용은 재귀적인 형태의 함수호출이 가능한 이유를 충분히 설명하고 있다. 실제로 함수를 구성하는 명령문은 cpu로 이동이 되어서 실행이 된다. 그런데 이 명령문은 얼마든지 cpu로 이동이 가능하다. 따라서 recursive 함수의 중간쯤 위치한 명령문을 실행하다가 다시 recursive 함수의 앞 부분에 위치한 명령문을 cpu로 이동시키는 것은 문제가 되지 않는다. 그래서 재귀적인 형태의 함수호출이 가능한 것이다.

탈출 조건을 추가해서 예제를 작성해보겠다.

```
#include <stdio.h>

void recur(int num)
{
        if(num<=0) // 재귀의 탈출조건
                return; // 재귀의 탈출!
        printf("Recursive call ! %d \n", num);
        recur(num-1);
}

int main(void)
{
        Recursive(3);
        return 0;
}
```

```
Recursive call ! 3
Recursive call ! 2
Recursive call ! 1
```

위의 예제에서 recur함수의 탈출조건에 해당하는 5 행을 보자. recur 함수에 전달되는 인자의 값이 0 이하인 경우에는 함수가 종료되도록 정의되어있다. 따라서 재귀적으로 호출이 이뤄지고 있는 recur 함수에 0 이 전달되면서 재귀의 탈출조건이 성립되어 함수가 반환하기 시작한다. 이렇듯 재귀함수를 정의하는데 있어서 탈출조건을 구성하는 것은 매우 중요한 일이다.