6주차 2차시 재귀함수

【학습목표】

1. 전역변수에 대해 설명할 수 있다.
2. 재귀함수에 대해 설명할 수 있다.

학습내용1 : 전역변수

1. 전역변수의 이해와 선언방법

```
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
```

√전역변수는 함수 외부에 선언된다.

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

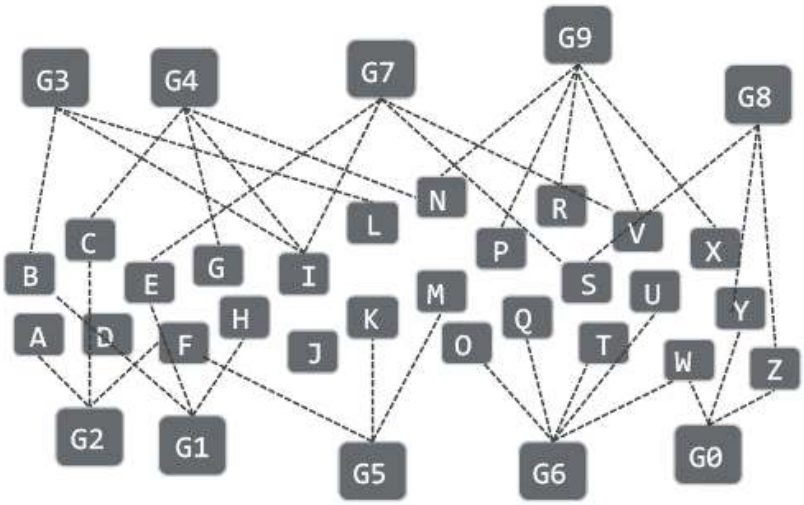
```
int Add(int val);
int num=1;

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

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

√지역변수의 이름이 전역변수의 이름을 가린다.

2. 전역변수! 많이 써도 되는가?



G0~G9의 전역변수와 함수와의 접근관계의 예시

√전역변수! 많이 쓰면 좋지 않다.
전역변수의 변경은 전체 프로그램의 변경으로 이어질 수 있으며 전역변수에 의존적인 코드는 프로그램 전체 영역에서 찾아야 한이다. 어디서든 접근이 가능한 변수이므로...

3. 지역변수에 static 선언을 추가한 static 변수

```
void SimpleFunc(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++)
        SimpleFunc();
    return 0;
}
```

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

√선언된 함수 내에서만 접근이 가능하다. (지역변수 특성)

√딱 1회 초기화되고 프로그램 종료 시까지 메모리 공간에 존재한다. (전역변수 특성)

√"난 사실 전역변수랑 성격이 같아.
초기화하지 않으면 전역변수처럼 0으로 초기화되고, 프로그램 시작과 동시에 할당 및
초기화되어서 프로그램이 종료될 때까지 메모리 공간에 남아있지!
그럼 왜 이 위치에 선언되었냐고?
그건 접근의 범위를 SimpleFunc로 제한하기 위해서야!"
static 지역변수의 발언!

√프로그램이 실행되면 static 지역변수는 해당 함수에 존재하지 않는다.

4. static 지역변수는 좀 써도 되나요?

√전역변수가 필요한 이유 중 하나는 다음과 같다.
선언된 변수가 함수를 빠져나가도 계속해서 메모리 공간에 존재할 필요가 있다.

√함수를 빠져나가도 계속해서 메모리 공간에 존재해야 하는 변수를 선언하는 방법은 다음 두 가지이다.
전역변수, static 지역 변수

√static 지역변수는 접근의 범위가 전역변수보다 훨씬 좁기 때문에 훨씬 안정적이다.
static 지역변수를 사용하여 전역변수의 선언을 최소화하자.

5. 보다 빠르게! register 변수

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

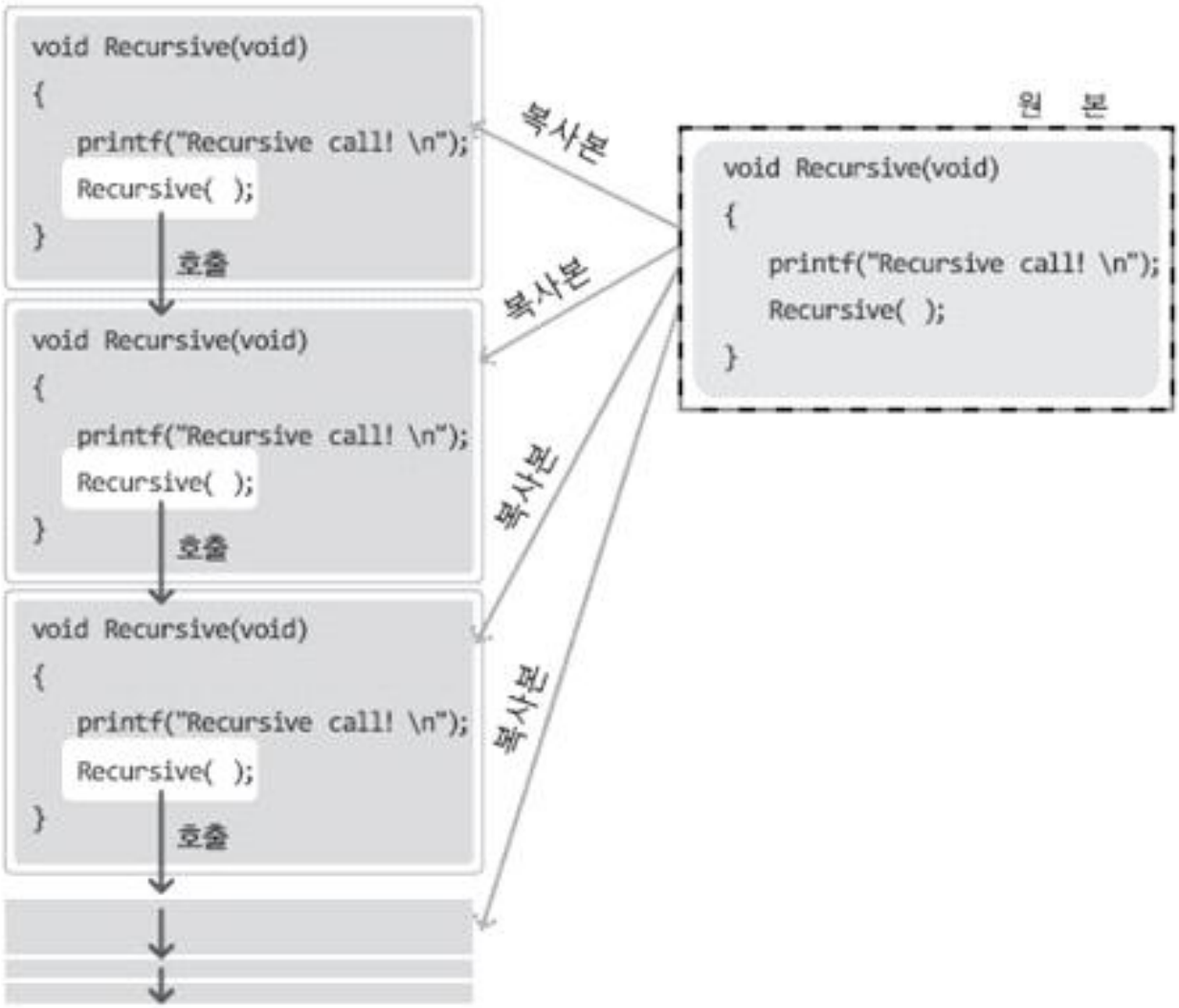
√register는 힌트를 제공하는 키워드이다. 컴파일러는 이를 무시하기도 한다.

그리고 레지스터는 CPU 내부에 존재하기 때문에 접근이 가장 빠른 메모리 장치이다.

√"이 변수는 내가 빈번히 사용하거든, 그래서 접근이 가장 빠른 레지스터에 저장하는 것이 성능향상에 도움이 될 거야"
register 변수 선언의 의미

학습내용2 : 재귀함수

1. 재귀함수의 기본적인 이해



재귀함수 호출의 이해!

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

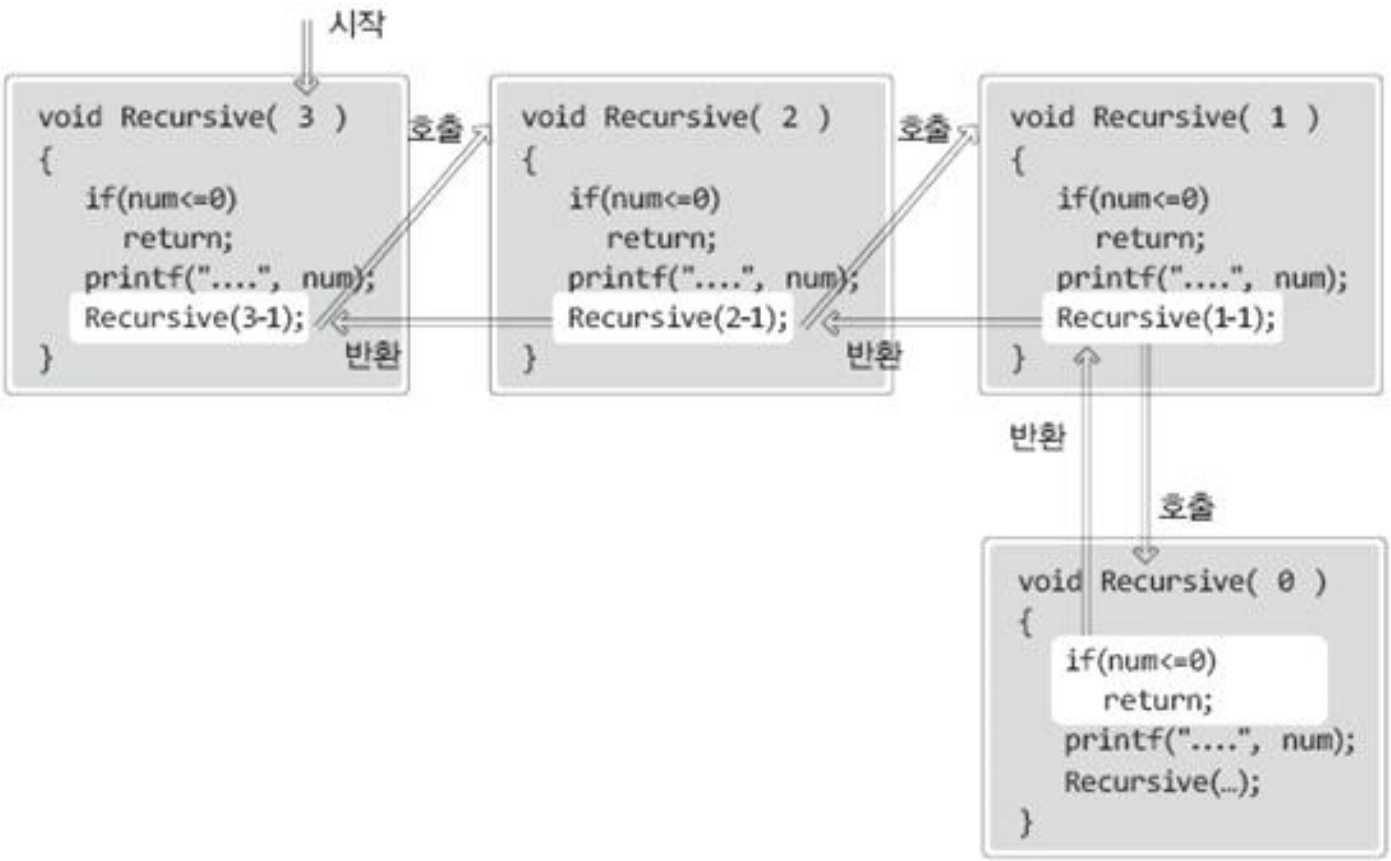
√자기자신을 재호출하는 형태로 정의된 함수를 가리켜 재귀함수라 한다.

2. 탈출조건이 존재하는 재귀함수의 예

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

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

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



√호출순서의 역순으로 반환이 이루어 진다.

3. 재귀함수의 디자인 사례

$n! = n \times \underbrace{(n-1) \times (n-2) \times (n-3) \times \cdots \times 2 \times 1}_{(n-1)!}$

$n! = n \times (n-1)!$

팩토리얼에 대한 수학적 표현

$$f(n) = \begin{cases} n \times f(n-1) & \cdots n \geq 1 \\ 1 & \cdots n = 0 \end{cases}$$

n x f(n-1) .... n>=1 에 대한 코드구현

```
if(n>=1)
    return n * Factorial(n-1);
```

f(n)=1 에 대한 코드구현

```
if(n==0)
    return 1;
```

```
if(n==0)
    return 1;
else
    return n * Factorial(n-1);
```

4. 팩토리얼 함수의 예

```
int Factorial(int n)
{
    if(n==0)
        return 1;
    else
        return n * Factorial(n-1);
}

int main(void)
{
    printf("1! = %d \n", Factorial(1));
    printf("2! = %d \n", Factorial(2));
    printf("3! = %d \n", Factorial(3));
    printf("4! = %d \n", Factorial(4));
    printf("9! = %d \n", Factorial(9));
    return 0;
}
```

```
1! = 1
2! = 2
3! = 6
4! = 24
9! = 362880
```

√C언어가 재귀적 함수 호출을 지원한다는 것은 그만큼 표현할 수 있는 범위가 넓다는 것을 의미한다.
√C언어의 재귀함수를 이용하면 재귀적으로 작성된 식을 그대로 코드로 옮길 수 있다.

【학습정리】

1. static 지역변수는 전역변수보다 안정적이다. 전역변수와 같이 프로그램이 종료될 때 까지 메모리 공간에 남아있지만 접근할 수 있는 범위를 하나의 함수로 제한했기 때문에 안정적이다.

2. 지역변수 앞에 register라는 선언을 추가하여 CPU내에 존재하는 레지스터라는 메모리 공간에 저장될 확률이 높아진다.

3. 전역변수 선언은 가급적 제한해야 한다. 왜냐하면 전역변수는 프로그램의 구조를 복잡하게 만드는 주범이기 때문이다.

4. 재귀함수에서 완료되지 않은 함수를 다시 호출하는 것이 가능하다.