

프로그래밍 1

Lecture Note #09

백윤철
ybaek@smu.ac.kr

내용

- 함수의 정의
- 지역변수
- 전역변수
- static
- register
- 재귀함수

함수를 만드는 이유

"Divide and Conquer!"

다수의 작은 단위 함수를 만들어서 프로그램을 작성하면 큰 문제를 작게 쪼개서 해결하는 효과를 얻을 수 있다. 그러나 함수를 만드는 이유 및 이점은 이보다 훨씬 다양하다.

- 코드의 재사용
- 중복되는 코드를 함수로 처리

- main 함수를 포함하여 함수의 크기는 작을수록 좋다. 무조건 작다고 좋은 것은 아니지만, 불필요하게 큰 함수가 만들어지지 않도록 주의해야 함
- 하나의 함수는 하나의 일만 담당하도록 디자인 되어야 한다. 물론 **하나의 일**이라는 것은 매우 주관적인 기준이다. 그러나 이러한 주관적 기준 역시 프로그래밍에 대한 경험이 쌓이면 매우 명확한 기준이 됨

함수의 생김새

```
반환형태 함수이름(argument선언)
{
    함수의 몸체
}
```

```
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;
}
```

printf 함수도 사실상 값을 반환한다. 다만 반환값이 필요 없어서 반환되는 값을 저장하지 않았을 뿐이다.
printf 함수는 출력된 문자열의 길이를 반환한다.

실행결과

12345

I love my home

6 15

함수의 입력과 출력

- 함수가 값을 반환하면, 반환된 값이 함수의 호출문을 대체한다고 생각하면 됨
- 예를 들어 아래의 printf() 함수 호출문이 6을 반환한다면,

```
num1 = printf("12345\n");
```

- 함수의 호출결과는 다음과 같이 되어 대입 연산이 진행됨


```
num1 = 6;
```

함수의 유형 구분

종류	argument	반환 값
유형 1	있음	있음
유형 2	있음	없음
유형 3	없음	있음
유형 4	없음	없음

유형 1. argument 있음. return value 있음

- Argument는 int형 정수 두 개이며 이 두 정수를 이용한 덧셈을 진행함
- 덧셈결과는 반환되며, 따라서 반환형도 int형으로 선언
- 함수의 이름은 Add



```
A. B. C.  
int Add (int num1, int num2)  
{  
    int result = num1 + num2;  
D. return result;  
}
```

- A. 반환형
- B. 함수의 이름
- C. 매개변수
- D. 값의 반환

유형 1. argument 있음. return value 있음

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

덧셈이 선 진행되고 그 결과가 반환됨

함수호출이 완료되면 호출한 위치로 이동해서 실행을 이어간다.

```
int main(void) {  
    int result;  
    result = Add(3, 4);  
    printf("덧셈결과1: %d\n", result);  
    result = Add(5, 8);  
    printf("덧셈결과2: %d\n", result);  
    return 0;  
}
```

실행결과

덧셈결과1: 7
덧셈결과2: 13

유형 2~3. argument 또는 return이 없음

```
/* 유형 2. 인자전달 (o), 반환 값 (x) */  
void ShowAddResult(int num) {  
    printf("덧셈결과 출력:%d\n", num);  
}
```

```
/* 유형 3 인자전달 (x), 반환 값 (o) */  
int ReadNum(void) {  
    int num;  
    scanf("%d", &num);  
    return num;  
}
```

유형 4. argument와 return이 모두 없음

```
void HowTouseThisProg(void) {  
    printf("두 개의 정수를 입력하면 덧셈 결과 출력\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;  
}
```

실행결과

두 개의 정수를 입력하시면 덧셈결과가 출력됩니다.
자! 그럼 두 개의 정수를 입력하세요.

12 24

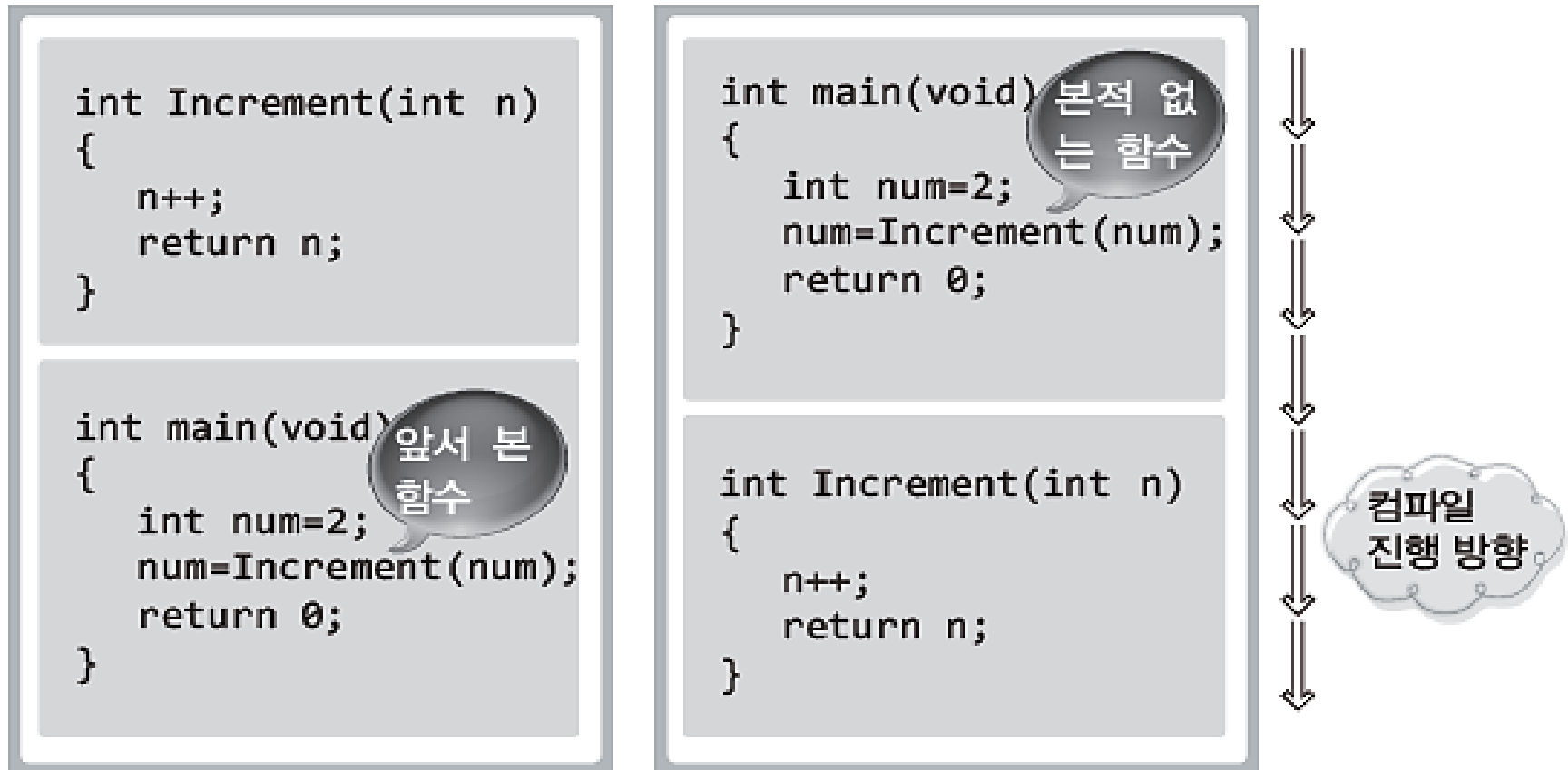
덧셈결과 출력: 36

값을 반환하지 않는 return 키워드

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

- return 문에는 '값의 반환'과 '함수의 탈출'이라는 두 가지 기능이 담겨 있음
- 위에서 보인 것처럼 값을 반환하지 않는 형태로, return 문을 구성하여 단순히 함수의 실행을 종료시키는 용도로 사용할 수 있음

함수의 정의와 그에 따른 원형의 선언



- ❑ 컴파일이 위에서 아래로 진행이 되기 때문에 함수의 배치순서는 중요하다. 컴파일 되지 않은 함수는 호출이 불가능하다.

함수의 정의와 그에 따른 원형의 선언

- 이후에 등장하는 함수에 대한 정보를 컴파일러에게 제공해서 이후에 등장하는 함수를 호출하는 문장이 컴파일될 수 있도록 함
- 이렇게 제공되는 함수의 정보를 가리켜 "함수의 선언" 이라고 함
- 함수의 선언은 보통
반환값 함수이름(argument 선언);
형태로 구성됨

```
int Increment(int n); /* 함수의 선언 */  
/* 위와 동일한 함수 선언. 매개변수 이름 생략 가능 */  
int Increment(int);
```

다양한 종류의 함수 정의1

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

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

중간에도 얼마든지 return문이 올 수 있다.

실행결과

3과 4중에서 큰 수는 4 이다.
7과 2중에서 큰 수는 7 이다.

다양한 종류의 함수 정의1

```
printf("3과 4중에서 큰 수는 %d\n",  
      Compare(3, 4));  
printf("7과 2중에서 큰 수는 %d\n",  
      Compare(7, 2));
```

위의 두 문장은 Compare
함수호출 이후 아래와 같이 된다.

```
printf("3과 4중에서 큰 수는 %d\n", 4);  
printf("7과 2중에서 큰 수는 %d\n", 7);
```

다양한 종류의 함수 정의2

```
/* 절댓값이 큰 정수 반환 */  
int AbsCompare(int n1, int n2);  
int GetAbsValue(int n); /* n의 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, AbsCompare(num1, num2));  
    return 0;  
}
```


다양한 종류의 함수 정의2

```
int AbsCompare(int n1, int n2) {  
    if (GetAbsValue(n1) > GetAbsValue(n2))  
        return n1;  
    else  
        return n2;  
}
```

이 예제에서 보이듯이 반환값이
있는 함수의 호출 문장은
표현식에서 사용 가능하다

```
int GetAbsValue(int n) {  
    if (n < 0)  
        return n * (-1);  
    else  
        return n;  
}
```

실행결과

두 개의 정수 입력: 5 -9
5와 -9중 절댓값이 큰 정수: -9

함수 내에서만 존재 및 접근 가능한 지역변수

- 함수 내에서 정의되는 변수를 지역변수(Local variable)라고 부름
- 지역변수는 정의된 이후로부터 함수 내에서만 접근이 가능함
- 한 지역(블록 영역) 내에 동일한 이름의 변수 정의 불가
 - 다른 지역에 동일한 이름의 변수 정의 가능
- 해당 지역(블록 영역)을 빠져나가면 지역변수는 소멸됨. 그리고 함수가 호출될 때마다 새롭게 할당됨

메모리 공간의 할당과 소멸 관찰하기

```
int SimpleFuncOne(void)
{
    int num=10;    // 이후부터 SimpleFuncOne의 num 유효
    num++;
    printf("SimpleFuncOne num: %d \n", num);
    return 0;    // SimpleFuncOne의 num이 유효한 마지막 문장
}
```

```
int SimpleFuncTwo(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 유효
    SimpleFuncOne();
    SimpleFuncTwo();
    printf("main num: %d \n", num);
    return 0;    // main의 num이 유효한 마지막 문장
}
```

함수 내에서만 존재 및 접근 가능한 지역변수

```
int SimpleFuncOne(void) {
    int num = 10; /* 이후부터 이 함수의 num 유효 */
    num++;
    printf("SimpleFuncOne num: %d\n", num);
    return 0; /* 이 함수의 num이 유효한 마지막 문장 */
}

int SimpleFuncTwo(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 */
}
```

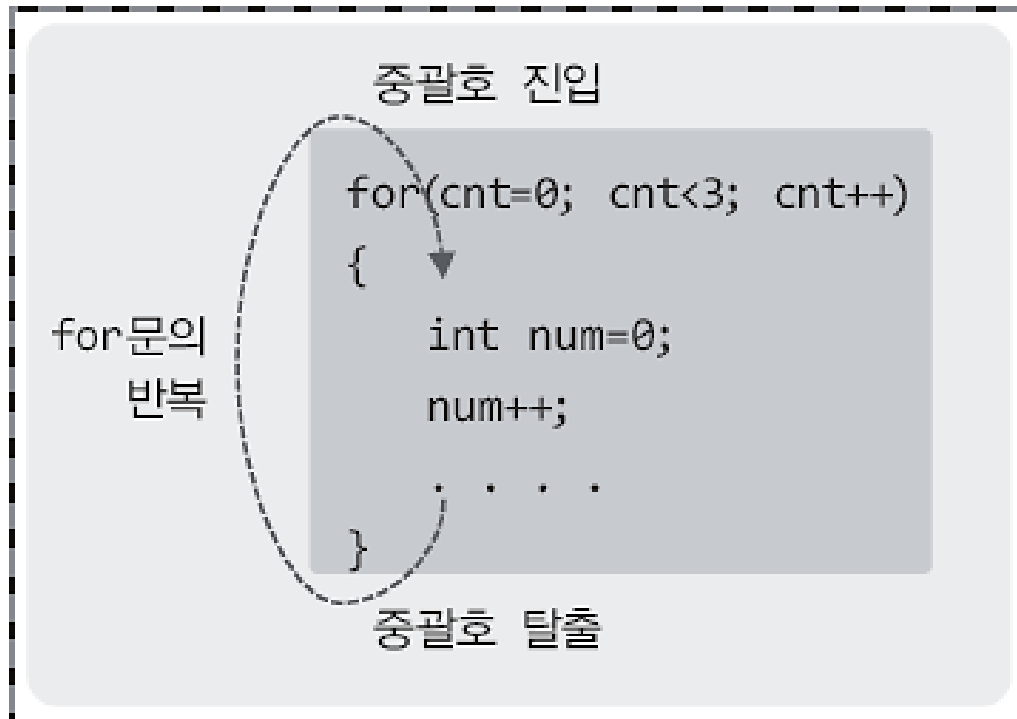
함수 내에서만 존재 및 접근 가능한 지역변수

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

실행결과

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

다양한 형태의 지역변수



- for문의 중괄호 내에 정의된 변수도 지역 변수
- for문의 중괄호를 빠져나가면 소멸됨

다양한 형태의 지역변수

- 지역변수는 외부에 정의된 동일한 이름의 변수를 가림

```
int main(void) { 실행결과
    int num = 1;
    if (num == 1) {
        int num = 7;
        num += 10;
        printf("if문 내 지역변수 num: %d\n", num);
    }
    printf("main 함수 내 지역변수 num: %d\n", num);
    return 0;
} 주석처리 후 실행결과
```

if문 내 지역변수 num: 17
main 함수 내 지역변수 num: 1

if문 내에 선언된 변수 num이 main 함수의 변수 num을 가린다.

if문 내 지역변수 num: 11
main 함수 내 지역변수 num: 11

지역변수의 일종인 매개변수

- 함수의 매개변수는 지역변수에 해당됨
- 매개변수도 정의된 함수 내에서만 접근 가능
- 정의된 함수가 종료되면, 지역변수와 함께 매개변수도 소멸됨

전역변수의 이해와 정의 방법

- 전역변수는 함수 외부에 정의됨
- 프로그램이 시작할 때 메모리 공간에 할당되고 종료 시까지 존재함
- 별도의 값으로 초기화하지 않으면 컴파일러에 의해 0으로 초기화 됨
- 프로그램 전체 영역 어디서든 접근 가능

전역변수의 이해와 정의 방법

```
void Add(int val);  
int num; /* 전역변수는 기본적으로 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 값을 val만큼 증가 */  
}
```

실행결과

num:	0
num:	3
num:	4

지역변수의 이해와 정의 방법

```
int Add(int val);  
int num = 1;  
int main(void) {  
    int num = 5;  
    printf("num: %d\n", Add(3) + 9);  
    printf("num: %d\n", num + 9);  
    return 0;  
}
```

지역변수의
이름이
지역변수의
이름을 가린다.

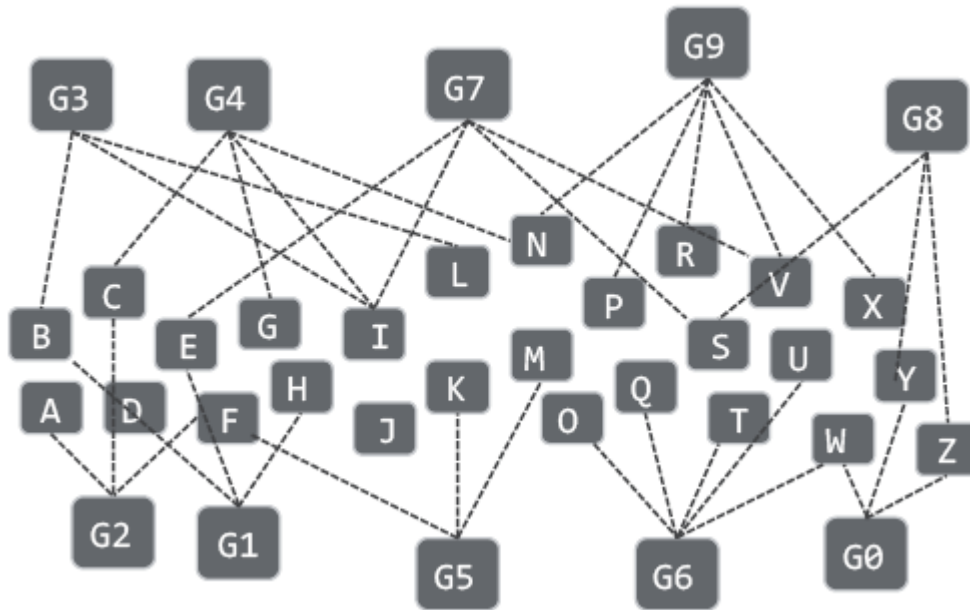
```
int Add(int val) {  
    /* int num = 0; */  
    num += val; /* 지역변수 num의 값을 val만큼 증가 */  
    return num;  
}
```

실행결과

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

num += val; /* 지역변수 num의 값을 val만큼 증가 */

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 한다. 어디서든 접근이 가능한 변수이므로...

static 변수

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

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

선언된 함수 내에서만 접근이
가능하다. (지역변수 특성)
딱 1회 초기화되고 프로그램 종료
시까지 메모리 공간에 존재한다.
(전역변수 특성)

실행결과

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

static 변수는 좀 써도 되나요?

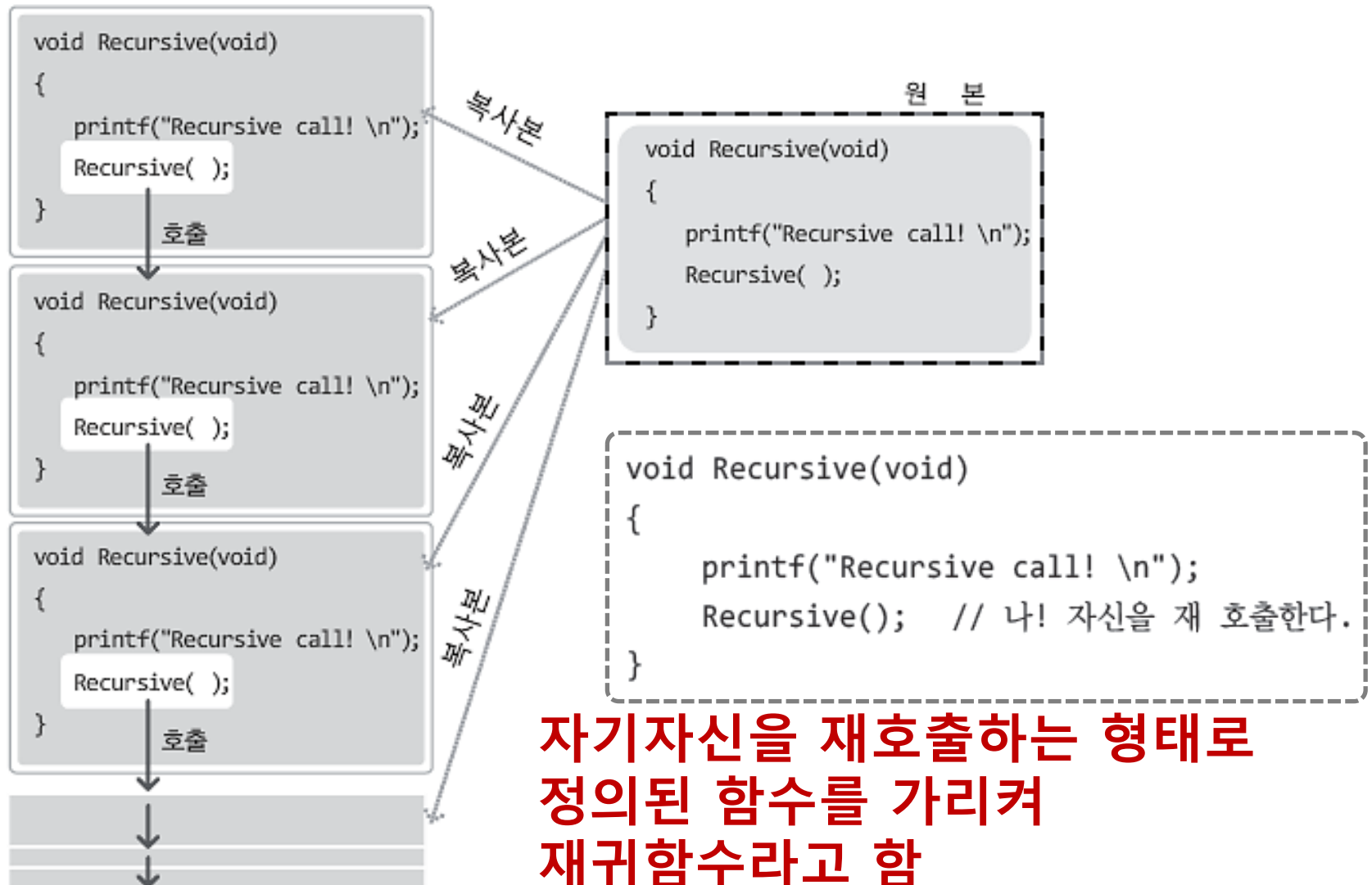
- 전역변수가 필요한 이유 중 하나는 다음과 같다
 - 정의된 변수가 함수를 빠져나가도 계속해서 메모리 공간에 존재할 필요가 있다 (값을 유지할 필요가 있다)
- 함수를 빠져나가도 계속해서 메모리 공간에 존재해야 하는 변수를 정의하는 방법은 다음 두 가지
 - 전역변수, static 변수
- static 변수는 접근의 범위가 전역변수보다 훨씬 좁기 때문에 안정적 (문제를 만들 소지가 전역변수에 비해 적음)
 - static 변수를 사용하여 전역변수의 선언을 최소화

보다 빠르게! register 변수

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

- register는 힌트를 제공하는 키워드. 컴파일러는 이를 무시하기도 함
- 레지스터는 CPU 내부에 존재하는, 때문에 가장 빠르게 동작하는 메모리 장치
- register 변수 선언의 의미
 - "이 변수는 내가 빈번히 사용하거든. 그래서 접근이 가장 빠른 레지스터에 저장하는 것이 성능향상에 도움될 거야"

재귀함수의 기본적 이해



탈출조건이 존재하는 재귀함수의 예

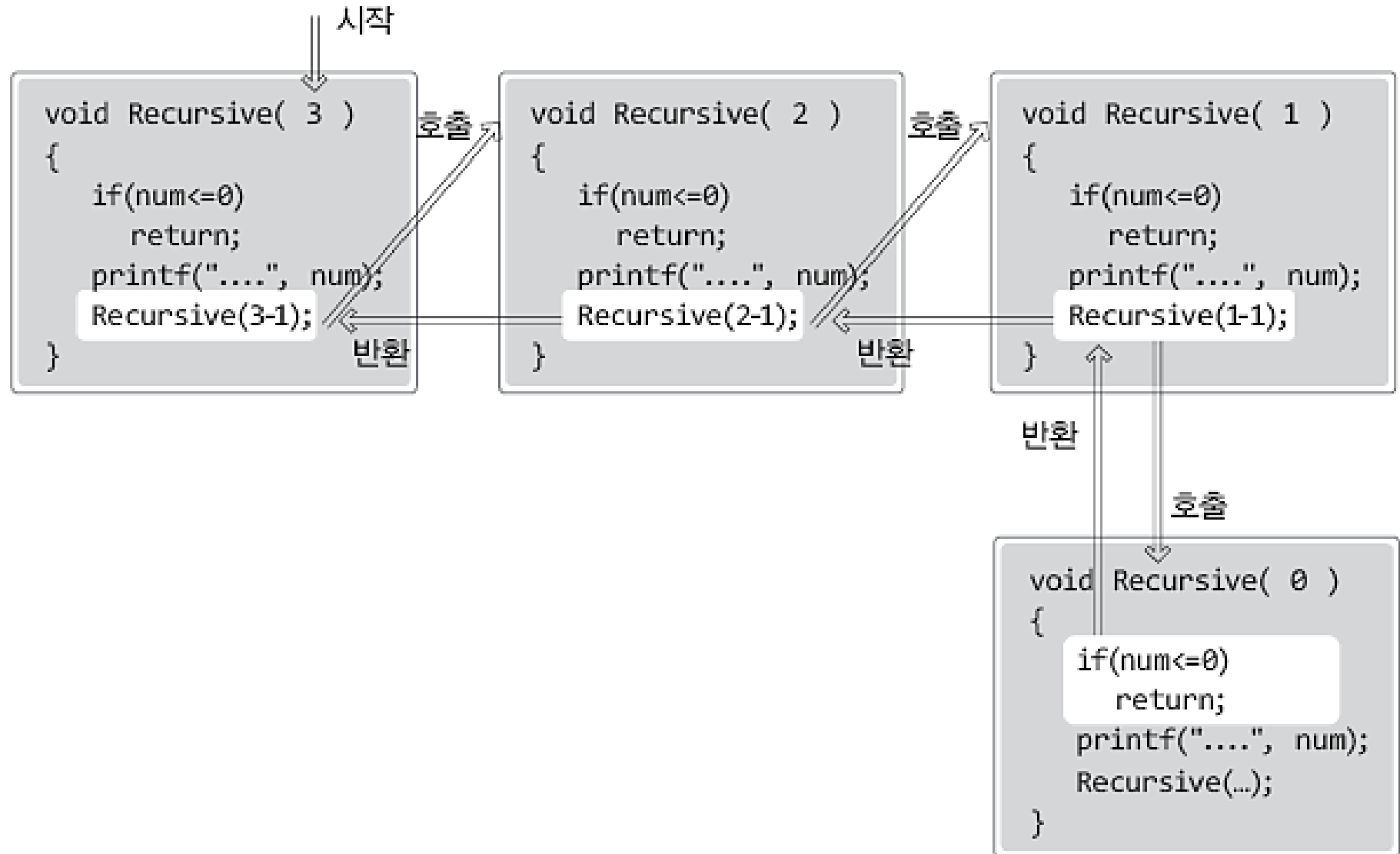
```
void Recursive(int num) {  
    if (num <= 0) /* 재귀 탈출 조건 */  
        return ;  
    printf("Recursive call! %d\n", num);  
    Recursive(num - 1);  
}
```

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

실행결과

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

탈출조건이 존재하는 재귀함수의 예



재귀 함수의 디자인 사례

$n! = n \times (n-1) \times (n-2) \times (n-3) \times \dots \times 2 \times 1$

$(n-1)!$



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



팩토리얼에 대한 수학적 표현

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



$n \times f(n-1) \dots n \geq 1$ 에 대한 코드 구현

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

$f(n)=1$ 에 대한 코드 구현

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



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

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;  
}
```

실행결과

1! = 1

2! = 2

3! = 6

4! = 24

9! = 362880

재귀 함수

- C언어가 재귀적 함수호출을 지원한다는 것은 그만큼 표현할 수 있는 범위가 넓다는 것을 의미
- C언어의 재귀함수를 이용하면 재귀적으로 작성된 식을 그대로 코드로 옮길 수 있음

정리

- 함수의 정의
- 지역변수
- 전역변수
- static
- register
- 재귀함수