IT프로그래밍

한성대학교
IT융합공학부
오희석
(ohhs@hansung.ac.kr)

Chapter 09-1. 함수를 정의하고 선언하기

Chapter 09. C언어의 핵심! 함수!

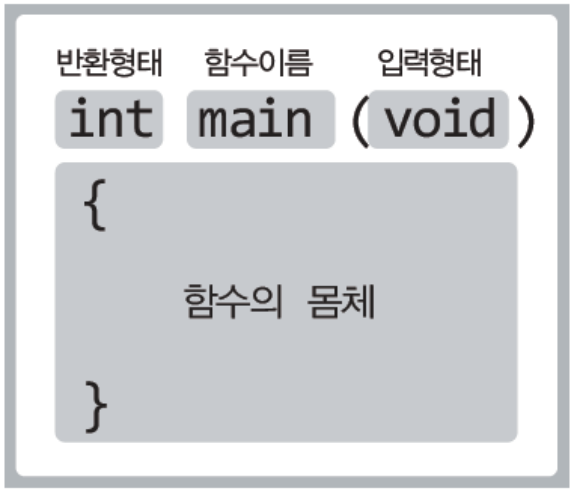
함수를 만드는 이유

"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.

main 함수를 포함하여 함수의 크기는 작을수록 좋다. 무조건 작다고 좋은 것은 아니지만, 불필요하게 큰 함수가 만들어지지 않도록 주의해야 한다.

하나의 함수는 하나의 일만 담당하도록 디자인 되어야 한다. 물론 하나의 일이라는 것은 매우 주관적인 기준이다. 그러나 이러한 주관적 기준 역시 프로그래밍에 대한 경험이 쌓이면 매우 명확한 기준이 된다.



함수의 입력과 출력: printf 함수도 반환을 합니다.

```
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.

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

실행결과

함수가 값을 반환하면 반환된 값이 함수의 호출문을 대체한다고 생각하면 된다.

예를 들어서 아래의 printf 함수 호출문이 6을 반환한다면,

num1=printf("12345\n");

함수의 호출결과는 다음과 같이 되어 대입연산이 진행된다.

num1=6;

함수의 구분

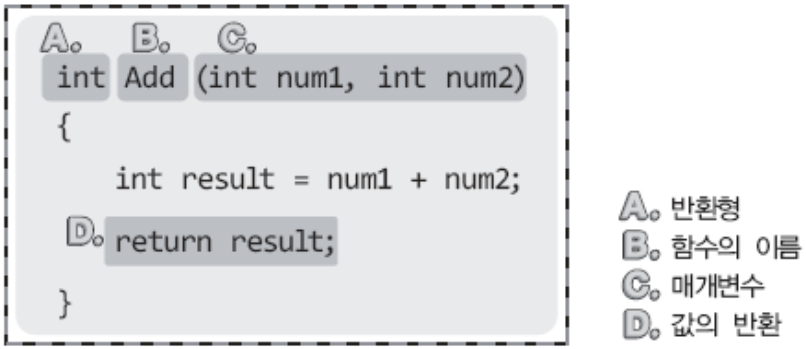
유형 1: 전달인자 있고, 반환 값 있다! 전달인자(○), 반환 값(○)

유형 2: 전달인자 있고, 반환 값 없다! 전달인자(○), 반환 값(×)

유형 3: 전달인자 없고, 반환 값 있다! 전달인자(×), 반환 값(○)

유형 4: 전달인자 없고, 반환 값 없다! 전달인자(×), 반환 값(×)

전달인자와 반환 값의 유무에 따른 함수의 구분!



전달인자 반환 값 모두 있는 경우

전달인자는 int형 정수 둘이며, 이 둘을 이용한 덧셈을 진행한다.

덧셈결과는 반환이 되며, 따라서 반환형도 int형으로 선언한다.

마지막으로 함수의 이름은 Add라 하자!

```
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

함수호출이 완료되면 호출한 위치로 이동해서 실행을 이어간다.

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

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;
}
```

덧셈이 선 진행되고 그 결과가 반환됨

실행결과

```
덧셈결과1: 7
덧셈결과2: 13
```

전달인자나 반환 값이 존재하지 않는 경우

```
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");
}
```

4가지 함수 유형을 조합한 예제

```
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;
}
```

실행결과

```
두 개의 정수를 입력하시면 덧셈결과가 출력됩니다.
자! 그럼 두 개의 정수를 입력하세요.
12 24
덧셈결과 출력: 36
```

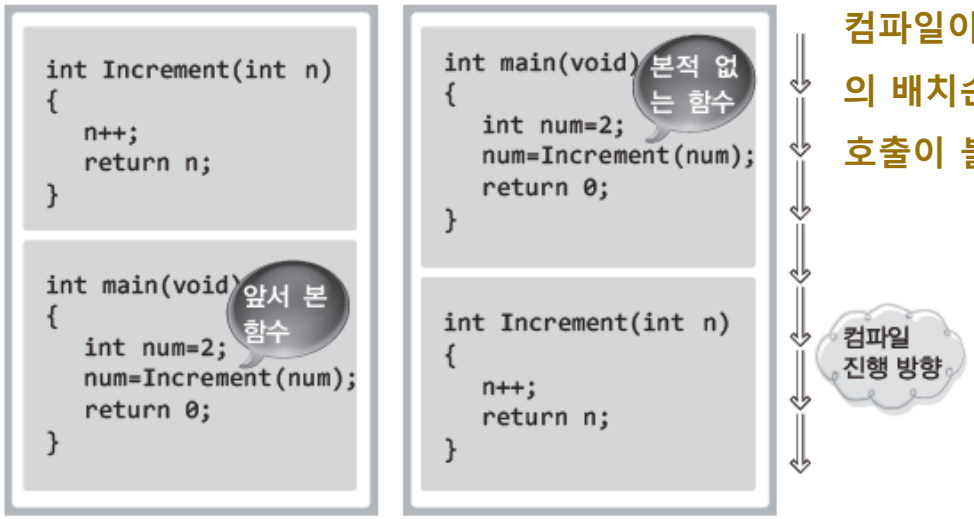
값을 반환하지 않는 return

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

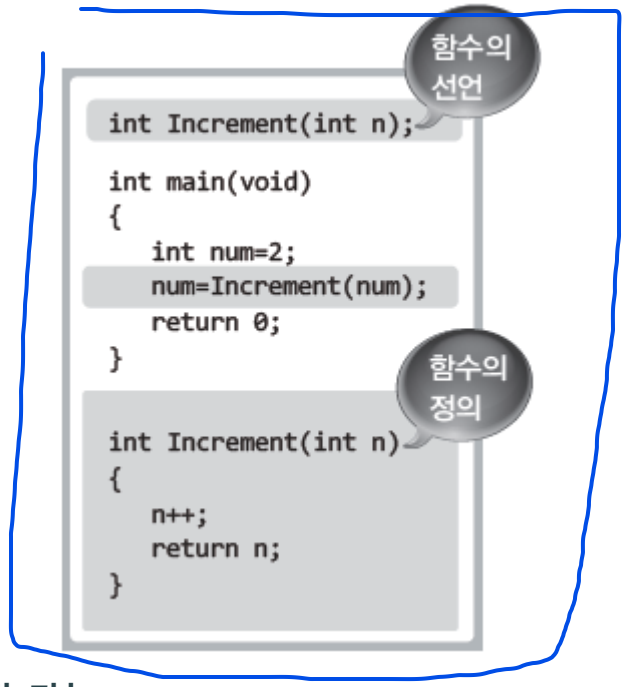
return문에는 '값의 반환'과 '함수의 탈출'이라는 두 가지 기능이 담겨있다.

위의 예제에서 보이듯이 값을 반환하지 않는 형태로 return문을 구성하여 값은 반환하지 않되 함수를 빠져나가는 용도로 사용할 수 있다.

함수의 정의와 그에 따른 원형의 선언



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



이후에 등장하는 함수에 대한 정보를 컴파일러에게 제공해서 이후에 등장하는 함수의 호출문장이 컴파일 가능하게 도울 수 있다.
이렇게 제공되는 함수의 정보를 가리켜 '함수의 선언'이라 한다.

```
int Increment(int n);   // 함수의 선언
int Increment(int);   // 위와 동일한 함수선언, 매개변수 이름 생략 가능
```

다양한 종류의 함수 정의1

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

int NumberCompare(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.
```

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

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

위의 두 문장한 NumberCompare 함수호출 이후 왼쪽과 같이 된다.

다양한 종류의 함수 정의2

```
int AbsoCompare(int num1, int num2);   // 절댓값이 큰 정수 반환
int GetAbsoValue(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, AbsoCompare(num1, num2));
    return 0;
}
int AbsoCompare(int num1, int num2)
{
    if(GetAbsoValue(num1) > GetAbsoValue(num2))
        return num1;
    else
        return num2;
}
int GetAbsoValue(int num)
{
    if(num<0)
        return num * (-1);
    else
        return num;
}
```

```
if(GetAbsoValue(num1) > GetAbsoValue(num2))
    . . . .

if( 5 > 9 )    GetAbsoValue 함수호출 이후
    . . . .
```

이 예제에서 보이듯이 함수의 호출문장은 어디에든 놓일 수 있다.

실행결과

```
두 개의 정수 입력: 5 -9
5와 -9중 절댓값이 큰 정수: -9
```

Chapter 09-2. 변수의 존재기간과 접근범위 1: 지역변수

Chapter 09. C언어의 핵심! 함수!

함수 내에만 존재 및 접근 가능한 지역변수

```
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 내에 선언되는 변수를 가리켜 지역변수라 한다.

지역변수는 선언된 이후로부터 함수 내에서만 접근이 가능하다.

한 지역(함수) 내에 동일한 이름의 변수 선언 불가능하다.

다른 지역에 동일한 이름의 변수 선언 가능하다.

해당 지역을 빠져나가면 지역변수는 소멸된다. 그리고 호출될 때마다 새롭게 할당된다.

실행결과

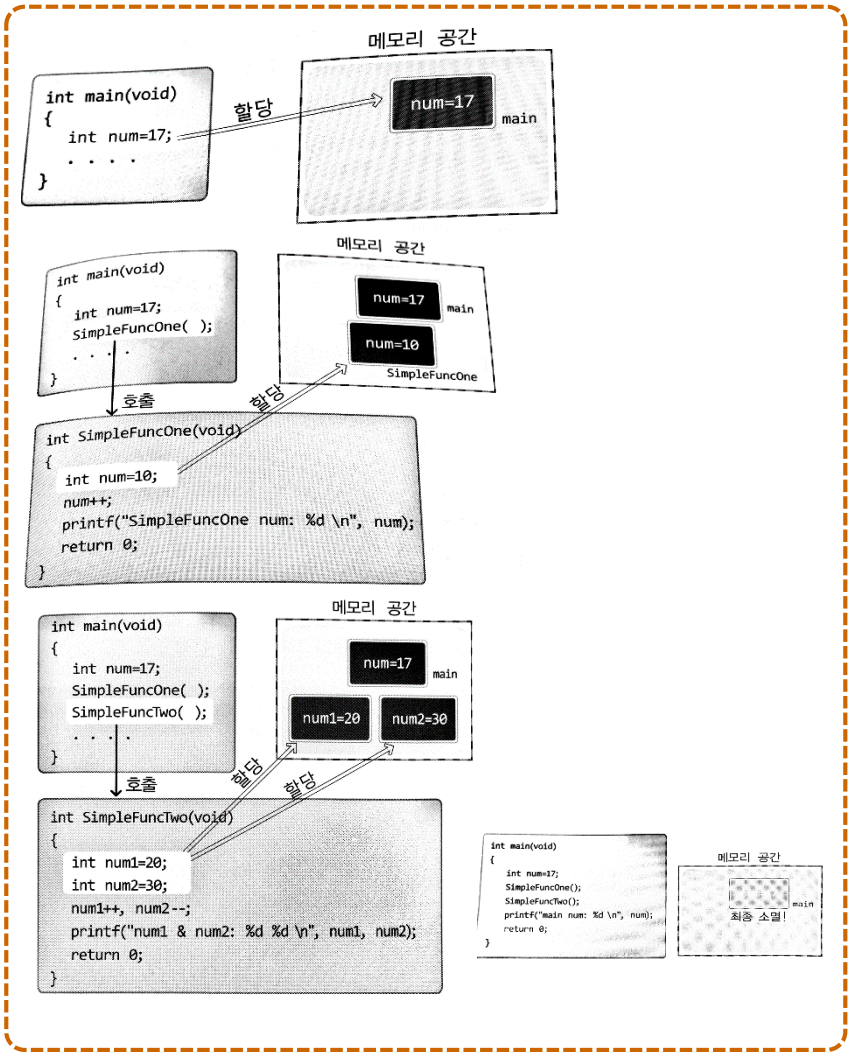
```
SimpleFuncOne num: 11
num1 & num2: 21 29
main num: 17
```

메모리 공간의 할당과 소멸 관찰하기

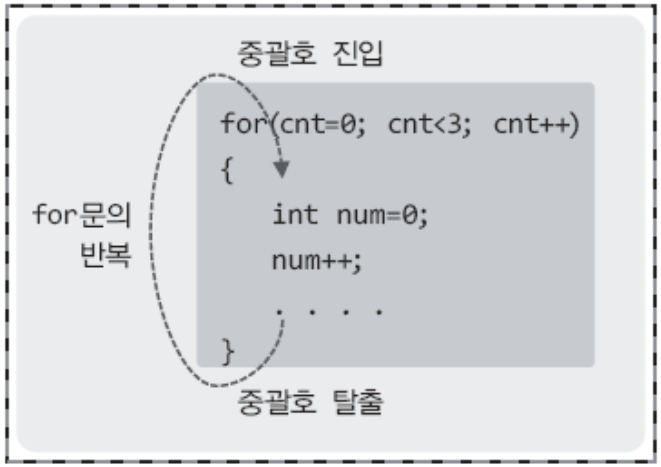
```
int SimpleFuncOne(void)
{
    int num=10;    // 이후부터 SimpleFuncOne의 num 유효
    num++;
    printf("SimpleFuncOne num: %d \n", num);
    return 0;    // SimpleFuncOne의 num이 유효한 마지막 문장
}

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
}
```



다양한 형태의 지역변수



for문의 중괄호 내에 선언된 변수도 지역변수이다. 그리고 이 지역변수는 for문의 중괄호를 빠져나가면 소멸된다.
따라서 for문의 반복횟수만큼 지역변수가 할당되고 소멸된다.

지역변수는 외부에 선언된 동일한 이름의 변수를 가린다.

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

if문 내에 선언된 변수 num이 main 함수의 변수 num을 가린다.

실행결과

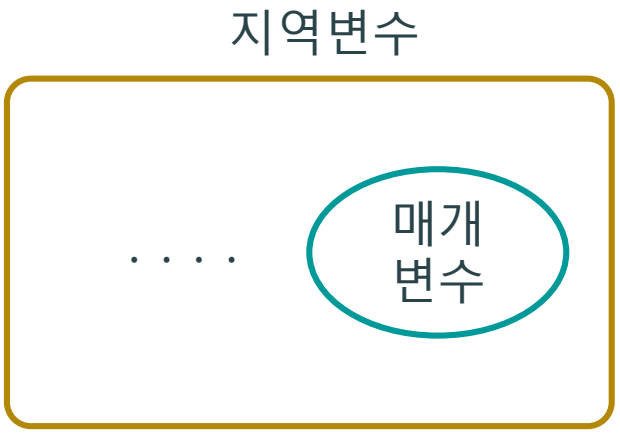
```
if문 내 지역변수 num: 17
main 함수 내 지역변수 num: 1
```

주석처리 후 실행결과

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

지역변수의 일종인 매개변수

int main (매개변수)



매개변수는 일종의 지역변수이다.

매개변수도 선언된 함수 내에서만 접근이 가능하다.

선언된 함수가 반환을 하면, 지역변수와 마찬가지로 매개변수도 소멸된다.

Chapter 09-3. 전역변수, static 변수, register 변수

Chapter 09. C언어의 핵심! 함수!

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
```

실행결과

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.

지역변수의 이름이
전역변수의 이름을 가린다.

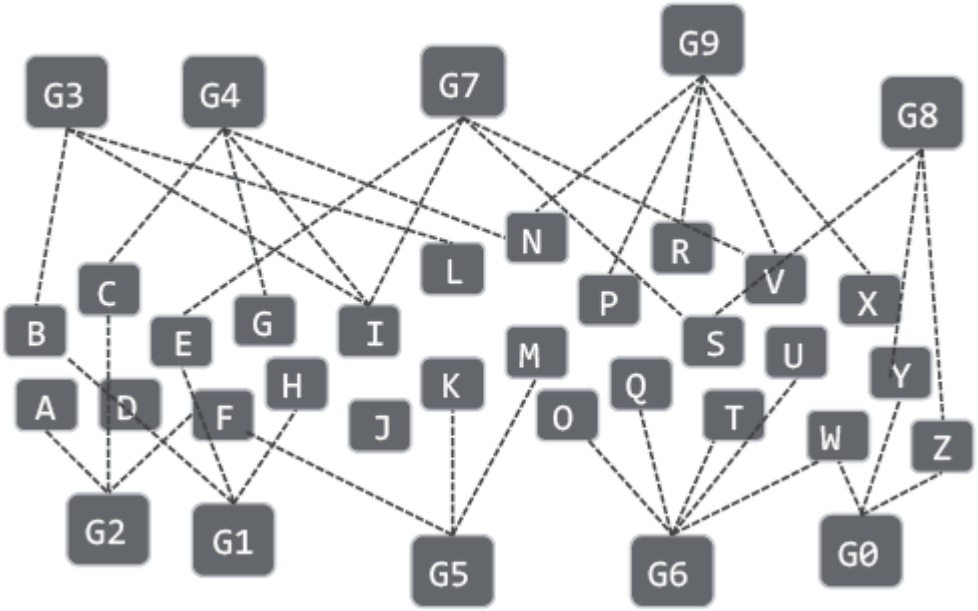
```
int Add(int val);
int num=1;

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;
}
```

실행결과

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

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...

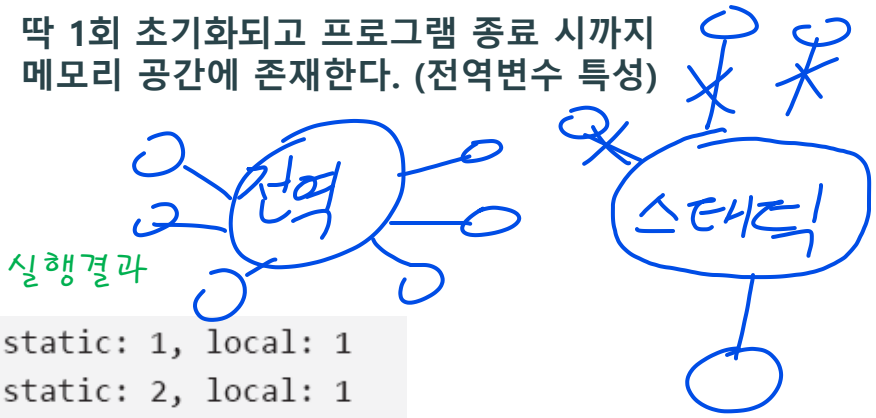
지역변수에 static 선언을 추가한 static 변수

```
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;
}
```

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
```

"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.

static 지역변수는 좀 써도 되나요?

ok

√ 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.

보다 빠르게! register 변수

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

register는 힌트를 제공하는 키워드이다. 컴파일러는 이를 무시하기도 한다.

그리고 레지스터는 CPU 내부에 존재하는, 때문에 접근이 가장 빠른 메모리 장치이다.

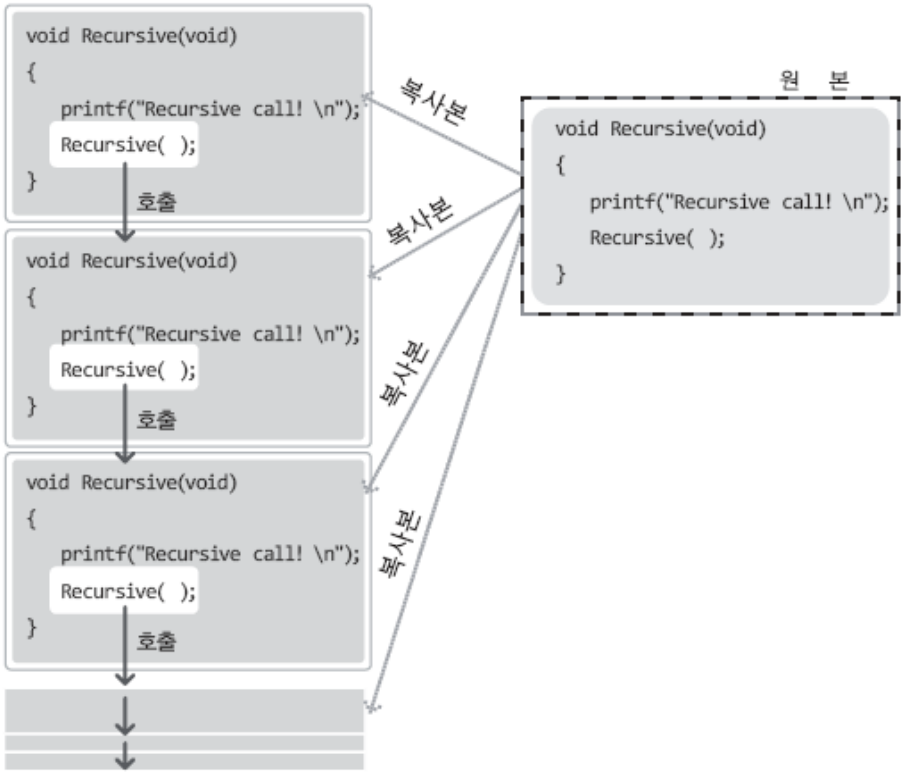
"이 변수는 내가 빈번히 사용하거든, 그래서 접근이 가장 빠른 레지스터에 저장하는 것이 성능향상에 도움이 될 거야"

register 변수 선언의 의미

Chapter 09-4. 재귀함수에 대한 이해

Chapter 09. C언어의 핵심! 함수!

재귀함수의 기본적인 이해

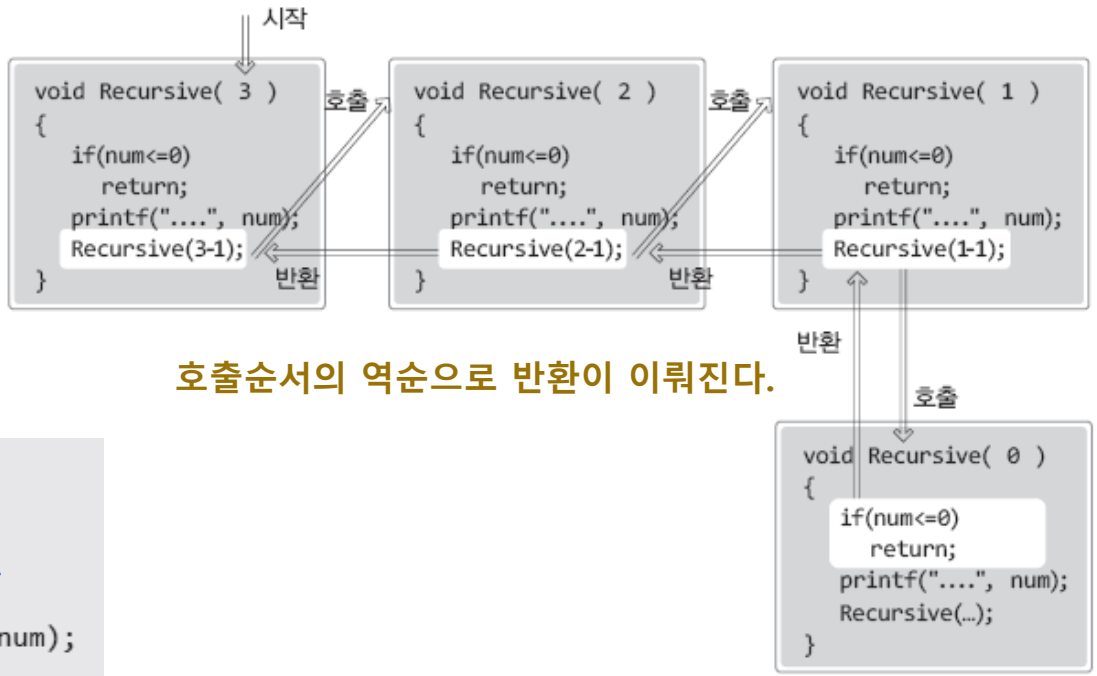


재귀함수 호출의 이해!

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

자기자신을 재호출하는 형태로 정의된 함수를 가리켜 재귀함수라 한다.

탈출조건이 존재하는 재귀함수의 예



호출순서의 역순으로 반환이 이뤄진다.

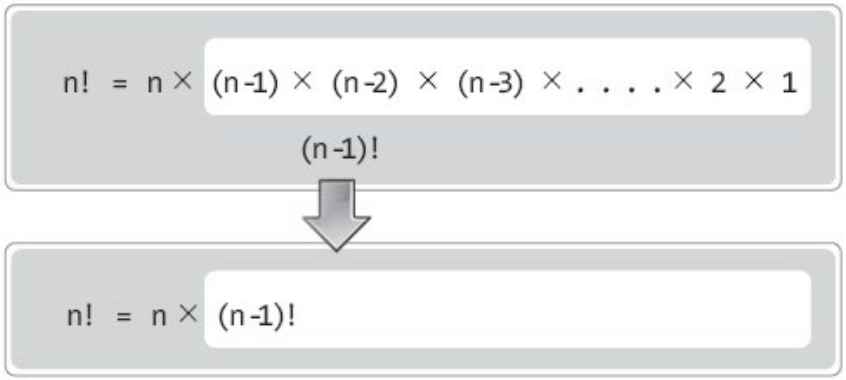
```
void Recursive(int num)
{
   if(num<=0)      // 재귀의 탈출조건
      return; //재귀의 탈출!
    printf("Recursive call! %d \n", num);
    Recursive(num-1);
}

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

실행결과

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

재귀함수의 디자인 사례



팩토리얼에 대한 수학적 표현

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

n x f(n-1) . . . . n>=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;
}
```

C언어가 재귀적 함수호출을 지원한다는 것은 그만큼 표현할 수 있는 범위가 넓다는 것을 의미한다!

C언어의 재귀함수를 이용하면 재귀적으로 작성된 식을 그대로 코드로 옮길 수 있다.

실행결과

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