
Introduction to Software Design

C03. Functions

Tae Hyun Kim

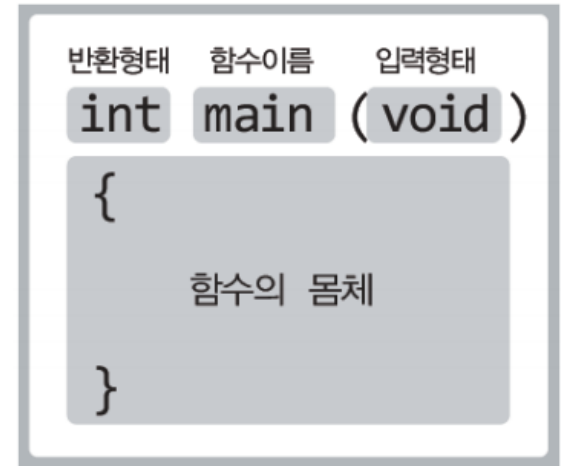
Topics Covered

- C 언어의 함수 (function)
 - 형태 및 위치
 - 왜 중요한가?
 - 실행 순서
- 변수의 범위 (scope)
 - 지역변수 (local variable)
 - 전역변수 (global variable)
- 재귀함수 (recursive function)

C 언어의 함수 (function)

- 예) 정수 두 개를 더한 후 그 더한 값을 반환하는 함수?

```
int add(int a, int b)
{
    return a + b;
}
```



Parameter & Return Value

- Parameter (매개 변수)
: 0개 or 1개 or 2개 or ... n개
- Return type (반환형) & Return value (반환 값)
: 있거나 or 없거나

```

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

함수 예제

```
int add(int a, int b)
{
    return a + b;
}
```

parameter 2개, return value 있음

```
void printAddResult(int a, int b)
{
    printf("result: %d\n", a + b);
}
```

parameter 2개, return value 없음
(void : "없다", "비었다"라는 의미)

```
void printHello() // void printHello(void)와 같음
{
    printf("hello world\n");
}
```

**parameter 0개,
return value 없음**
(parameter가 없을 땐
void를 써도 되고 아무
것도 안 써도 됨)

```
int scale2x(int a)
{
    return 2 * a;
}
```

parameter 1개, return value 있음

C & Python Examples

- C

```
#include <stdio.h>

int add(int a, int b)
{
    return a + b;
}

int main(void)
{
    int num1 = 1, num2 = 2;
    int num3 = add(num1, num2);

    printf("%d\n", num3);
    printf("%d\n", add(num1, num2));
    printf("%d\n", add(3, 8));

    return 0;
}
```

- Python

```
def add(a, b):
    return a + b

num1 = 1
num2 = 2
num3 = add(num1, num2)

print(num3)
print(add(num1, num2))
print(add(3, 8))
```

return

- return: 함수를 빠져나가면서 값을 반환
- 반환 값이 없는 함수는?

```
void printHello()  
{  
    printf("hello world\n");  
    // return 생략 가능  
}
```

=

```
void printHello()  
{  
    printf("hello world\n");  
    return; // return을 쓸 수도 있음  
}
```

return

- return이 반드시 함수의 마지막에 위치할 필요는 없다.

```
void printEvenNumber(int num)
{
    if(num%2 == 1)
        return;

    printf("Even number\n");
}
```

```
int scale2xEvenNumber(int num)
{
    if(num%2 == 0)
        return num * 2;
    else
        return -1;
}
```


왜 함수를 잘 만드는 것이 중요한가?

- 프로그래밍은 문제를 해결하는 과정
- 문제가 복잡할 때?

Divide and Conquer!

- 복잡한 문제를 한꺼번에 해결하려는 것보다는, 작은 문제 여러 개로 나누어 하나씩 해결하는 것이 효과적
- 하나의 함수가 하나의 작은 문제를 담당하도록 설계
- 또 하나 중요한 점은, 함수를 사용하여 코드의 **중복 작성**을 피할 수 있음

함수를 이용해 코드의 중복작성을 피할 수 있는 예

```
#include <stdio.h>

int main(void)
{
    int num1, num2;

    printf("Enter a number: ");
    scanf("%d", &num1);
    printf("Entered number is %d\n", num1);

    if(num1%2==0)
    {
        printf("Enter a number: ");
        scanf("%d", &num2);
        printf("Entered number is %d\n", num2);

        printf("num1 + num2 = %d\n", num1 + num2);
    }
    else
    {
        num1 *= 2;

        printf("Enter a number: ");
        scanf("%d", &num2);
        printf("Entered number is %d\n", num2);

        printf("num1 - num2 = %d\n", num1 - num2);
    }

    return 0;
}
```

```
#include <stdio.h>

int getNumber()
{
    int num;

    printf("Enter a number: ");
    scanf("%d", &num);
    printf("Entered number is %d\n", num);

    return num;
}

int main(void)
{
    int num1, num2;
    num1 = getNumber();

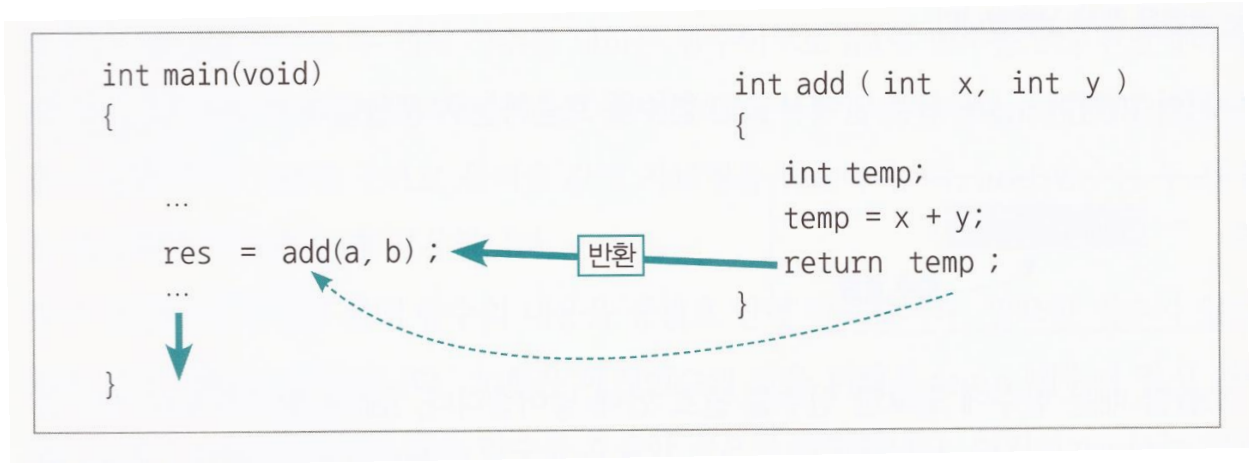
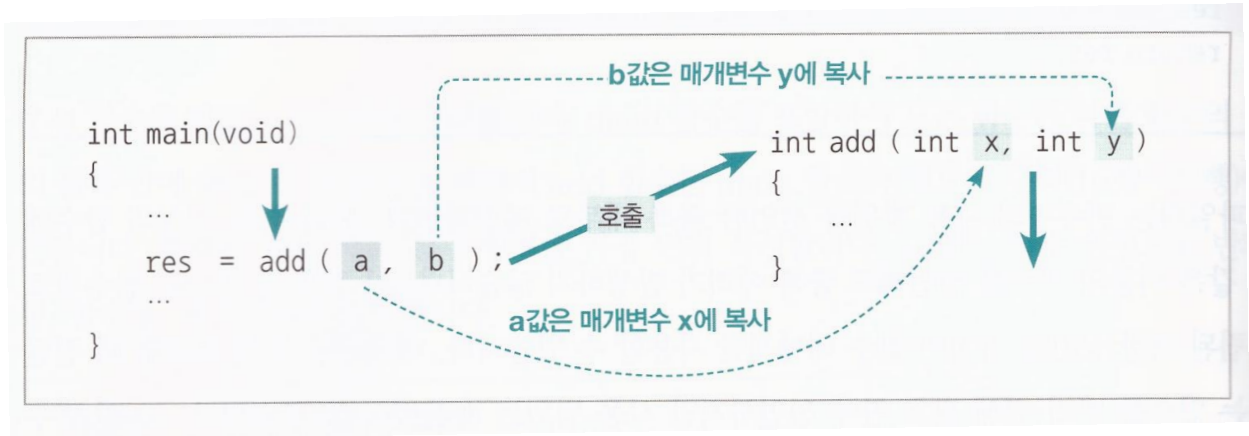
    if(num1%2==0)
    {
        num2 = getNumber();
        printf("num1 + num2 = %d\n", num1 + num2);
    }
    else
    {
        num1 *= 2;
        num2 = getNumber();
        printf("num1 - num2 = %d\n", num1 - num2);
    }

    return 0;
}
```

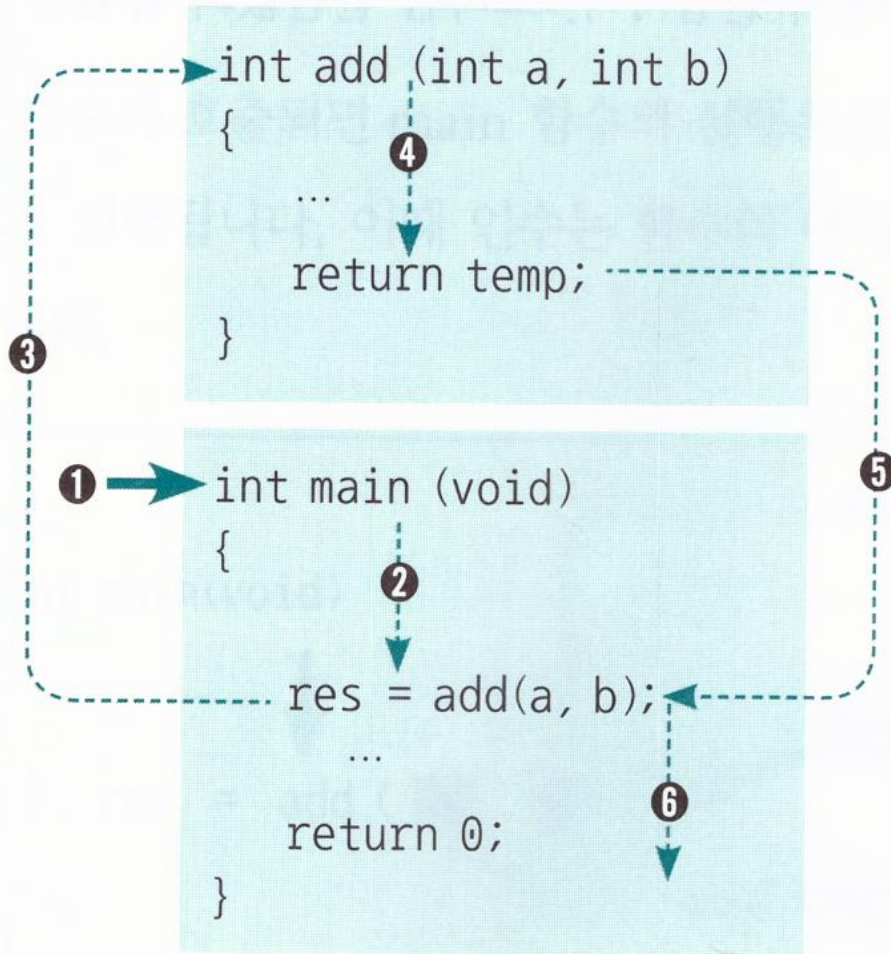
함수 호출 시 코드의 실행 순서

```
int add(int x, int y)
{
    int temp;
    temp = x + y;
    return temp;
}
```

```
int main()
{
    int a = 2, b = 5;
    int res = add(a, b);
    printf("%d\n", res);
    return 0;
}
```



함수 호출 시 코드의 실행 순서



- ❶ main 함수 시작
- ❷ main 함수 실행
- ❸ add 함수 호출
- ❹ add 함수 실행
- ❺ add 함수 반환
- ❻ 나머지 main 함수 실행

함수의 소스코드에서의 위치는?

- 어느 쪽이 맞는 코드?

```
// test1.c
#include <stdio.h>

int add(int a, int b)
{
    return a + b;
}

int main(void)
{
    int num;
    num = add(3, 8);
    printf("%d\n", num);

    return 0;
}
```

앞에서 본 함수

```
// test2.c
#include <stdio.h>

int main(void)
{
    int num;
    num = add(3, 8);
    printf("%d\n", num);

    return 0;
}

int add(int a, int b)
{
    return a + b;
}
```

본 적 없는 함수

컴파일
진행방향

함수의 선언

- 반드시 모든 함수가 사용되기 전에 정의되어야 하는가?
-> 그렇지 **않다!** 무슨 함수를 쓸지만 미리 알려 주면 된다.

```
// test2.c
#include <stdio.h>

int add(int a, int b);

int main(void)
{
    int num;
    num = add(3, 8);
    printf("%d\n", num);

    return 0;
}
```

함수의 선언 (declaration)

: "내가 어떤 함수를 쓰겠다"라고 선언하는 것
(컴파일러에게 미리 알려줌)

```
int add(int a, int b)
{
    return a + b;
}
```

함수의 정의 (definition)

: 실제 그 함수를 만드는 것

참고) 함수의 선언만 제대로 하면
함수의 정의는 다른 파일에 작성 가능

C & Python Examples

- C

```
#include <stdio.h>

int add(int a, int b);

int main(void)
{
    int num;
    num = add(3, 8);
    printf("%d\n", num);

    return 0;
}

int add(int a, int b)
{
    return a + b;
}
```

- Python

```
# Python은 함수 호출 시점에 해당
# 함수의 정의를 알고 있으면 된다.
# 따라서 이 코드는 에러가 발생하지만,
num = add(3, 8)
print(num)
```

```
def add(a, b):
    return a + b
```

```
# 이 코드는 문제없이 실행된다
```

```
def main():
    num = add(3, 8)
    print(num)
```

```
def add(a, b):
    return a + b
```

```
main()
```

지역변수, 전역변수

- 변수는 얼마 동안 살아있을까?

-> 특정 변수가 메모리상에 존재하는 기간

- 코드의 어떤 부분에서 각 변수를 쓸 수 있을까?


-> 코드 내에서 특정 변수에 접근할 수 있는 범위

- 위 기준에 따라 크게 두 가지로 분류: **지역변수, 전역변수**

지역변수(Local Variable)

- 특정 범위 안에 선언되는 변수 (범위: 중괄호 안)


```
int main(void)
{
    int num = 1;
    printf("%d\n", num);
    return 0;
}
```



- 프로그램의 실행이 해당 범위를 지날 때 메모리상에 존재함
- 해당 범위 안에서만 접근 가능함


지역변수

```
int add(int a, int b)
{
    int c;
    c = a + b;
    return c;
}
```



- 함수의 parameter도 지역변수
- 존재 및 접근 가능 범위는 해당 함수

```
for (int i = 0; i < 5; i++)
{
    printf("%d ", i);
}
```



- 반복문 안에서 선언된 변수도 지역변수
- 존재 및 접근 가능 범위는 해당 반복 영역

```
int main()
{
    int num1;
    scanf("%d", &num1);

    if(num1 > 0)
    {
        int num2;
        num2 = num1 * 2;
        printf("%d\n", num2);
    }
    else
    {
        int num3;
        num3 = num1 / 2;
        printf("%d\n", num3);
    }

    return 0;
}
```

num2의 존재 및
접근 가능 범위

num3의 존재 및
접근 가능 범위

num1의 존재 및
접근 가능 범위

```
int main()
{
    int num1;
    scanf("%d", &num1);

    if(num1 > 0)
    {
        int num2;
        num2 = num1 * 2;
        printf("%d\n", num2);
    }
    else
    {
        int num3;
        num3 = num1 / 2;
        printf("%d\n", num3);
    }

    return 0;
}
```

메모리 공간에 num1 생성

메모리 공간에 num2 생성

메모리 공간에서 num2 소멸

메모리 공간에 num3 생성

메모리 공간에서 num3 소멸

메모리 공간에서 num1 소멸

C & Python Examples

- C

```
#include <stdio.h>

int add(int a, int b)
{
    return a + b;
}

int main(void)
{
    int num;
    num = add(3, 8);
    printf("%d\n", num);
    printf("%d\n", a); // error

    return 0;
}
```

- Python

```
def add(a, b):
    return a + b

num = add(3, 8)
print(num)
print(a) # error
```

전역변수(Global Variable)

- 함수 밖에 선언되는 변수

```
int gNum;  
  
int main()  
{  
    printf("%d\n", gNum);  
    return 0;  
}
```

- 프로그램이 처음 실행될 때부터 종료될 때까지 메모리상에 존재함
- 프로그램 전체 영역에서 접근 가능함

전역변수(Global Variable)



```
#include <stdio.h>

int gNum = 10;

void add(int a)
{
    gNum += a;
}

int main()
{
    printf("%d\n", gNum);
    add(3);
    printf("%d\n", gNum);
    gNum += 2;
    printf("%d\n", gNum);

    return 0;
}
```

- 프로그램이 처음 실행될 때 메모리 공간에 생성, 프로그램이 종료될 때 메모리 공간에서 소멸.
- 프로그램 전체 영역에서 접근 가능함

C & Python Examples

- C

```
#include <stdio.h>

int gNum = 10;

void add(int a)
{
    gNum += a;
}

int main()
{
    printf("%d\n", gNum);
    add(3);
    printf("%d\n", gNum);
    gNum += 2;
    printf("%d\n", gNum);

    return 0;
}
```

- Python

```
gNum = 10

def add(a):
    global gNum
    gNum += a

print(gNum)
add(3)
print(gNum)
add(2)
print(gNum)
```

- Python에서 특정 함수 내에게 global variable에 쓰기 접근을 하려면, 반드시 해당 함수에서 global 선언을 해주어야 한다.

전역변수의 사용에 대해

- 전역변수는 중간에 소멸되지도 않고 코드의 어디에서든 접근이 가능하니 편하게 많이 쓰면 되겠다!
- **절대 그렇게 하면 안 됨!!**
- 문제점:
 - 변수 범위가 넓어서 이해하기가 어려움.
 - 특정 전역변수가 어떻게 쓰이는지 파악하려면 프로그램 전체 코드를 살펴봐야 함. (지역변수는 변수가 선언된 범위만 살펴보면 됨)
 - 전역변수를 사용하는 특정 코드를 수정 -> 해당 전역변수를 사용하는 (눈에 금방 띄지 않는) 다른 모든 코드에 영향 -> **버그 가능성↑**

전역변수의 사용에 대해

- 전역변수를 많이 사용하는 코드는 이해하기 어렵고, 디버깅도 어렵다.
- 전역변수 사용이 필요한 경우: 프로그램에 하나만 존재하는 관리자(manager) 성격의 객체.
 - 예: 게임의 resource manager. 게임 진행 상황에 따라 로드된 특정 이미지, 모델, 사운드 등의 데이터를 관리함.
- 전역변수는 꼭 필요한 경우가 아니면 쓰지말 것!!

static 지역 변수

- 지역변수에 static 선언을 해주면

```
static int num1;
```

- 선언된 범위 안에서만 접근 가능함
 - > 지역변수 특성
- 프로그램이 처음 실행될 때부터 종료될 때까지 메모리상에 존재함
 - > 전역변수 특성

static 지역 변수

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

```
void addOneAndPrint()
```

```
{
```

```
    static int num1=0;
```

```
    int num2 = 0;
```

```
    num1++;
```

```
    num2++;
```

```
    printf("static: %d, local: %d\n", num1, num2);
```

```
%d\n", num1, num2);
```

```
}
```

```
int main()
```

```
{
```

```
    int i;
```

```
    for(i=0; i<3; i++)
```

```
        addOneAndPrint();
```

```
    return 0;
```

```
}
```

num1의
접근 가능 범위

num1은 프로그램이
처음 실행될 때부터
종료될 때까지
메모리상에 존재함

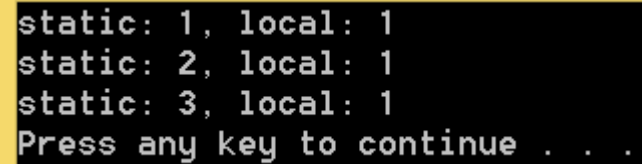
C Examples

- C

```
#include <stdio.h>

void addOneAndPrint ()
{
    static int num1=0;
    int num2 = 0;
    num1++;
    num2++;
    printf("static: %d,
local: %d\n", num1, num2);
}

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



```
static: 1, local: 1
static: 2, local: 1
static: 3, local: 1
Press any key to continue . . .
```

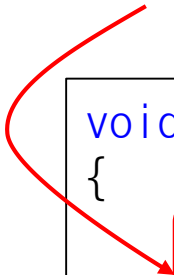
- Python에는 static local variable의 개념이 없다.

재귀 함수(Recursive Function)

- 완료되지 않은 함수를 호출하는 것이 가능한가요?
- 가능하다! 함수의 호출이란 것은 프로그램의 실행 위치가 함수의 시작 위치로 점프하는 것을 의미
- 함수의 리턴(return)은 프로그램의 실행 위치가 함수 실행 전 원래 위치로 복귀하는 것을 의미

재귀함수의 탈출조건

- 재귀함수를 제대로 쓰려면 **탈출조건**이 있어야 한다.
 - 탈출조건에서는 자기 자신을 다시 호출하지 않고 리턴해야 함



```
void recursive(int n)
{
    if(n==0)
        return;
    printf("recursive\n");
    recursive(n-1);
}

int main()
{
    recursive(3);
    return 0;
}
```

실행결과

```
recursive
recursive
recursive
```

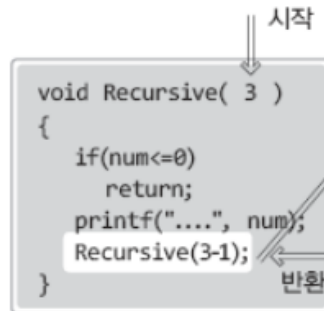
```
printf("recursive\n");
-> printf("recursive%d\n ", n);
//test 해볼 것!
```


재귀함수의 호출과정

```
void Recursive(int n)
{
    if(n==0)
        return;

    printf("recursive\n");
    Recursive(n-1);
}

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

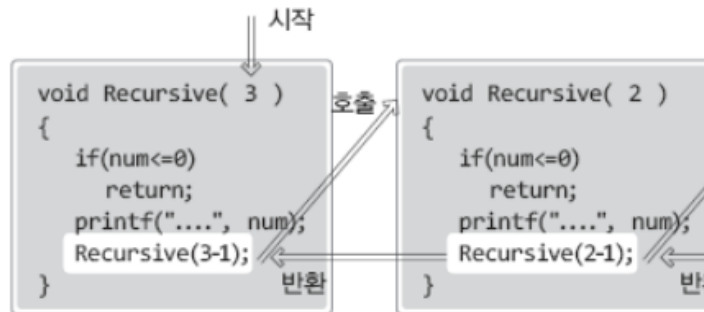


재귀함수의 호출과정

```
void Recursive(int n)
{
    if(n==0)
        return;

    printf("recursive\n");
    Recursive(n-1);
}

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

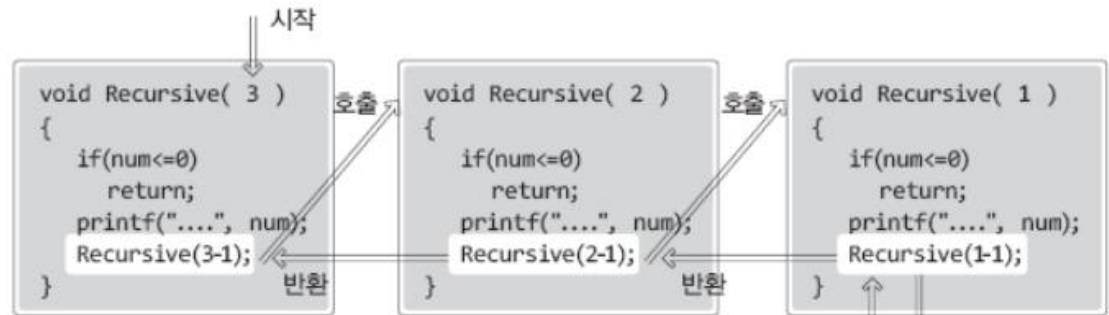


재귀함수의 호출과정

```
void Recursive(int n)
{
    if(n==0)
        return;

    printf("recursive\n");
    Recursive(n-1);
}

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

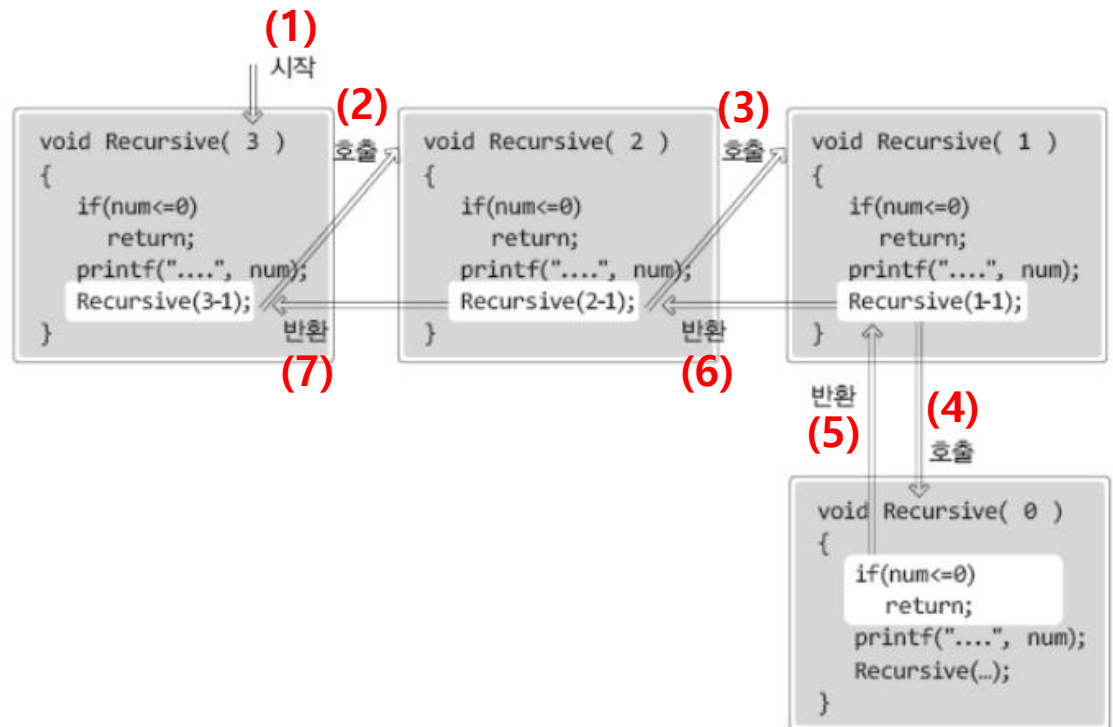


재귀함수의 호출과정

```
void Recursive(int n)
{
    if(n==0)
        return;

    printf("recursive\n");
    Recursive(n-1);
}

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



실행의 순서: (1) - (2) - (3) - (4) - (5) - (6) - (7)

C & Python Examples

- C

```
#include <stdio.h>

void recursive(int n)
{
    if(n==0)
        return;

    printf("before %d\n", n);
    recursive(n-1);
    printf("after %d\n", n);
}

int main()
{
    recursive(3);
    return 0;
}
```

- Python

```
def recursive(n):
    if n==0:
        return

    print('before %d'%n)
    recursive(n-1)
    print('after %d'%n)

recursive(3)
```

재귀함수를 쓰기 좋은 경우

- 점화식으로 표현되는 경우

- 예: 계승(Factorial)

$$- 5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

$$- F_n := \begin{cases} 1 & \text{if } n = 1; \\ n \times F_{n-1} & \text{if } n > 1. \end{cases}$$

- 예: 피보나치 수열

$$- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$$

$$- F_n := \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$

Example - Factorial

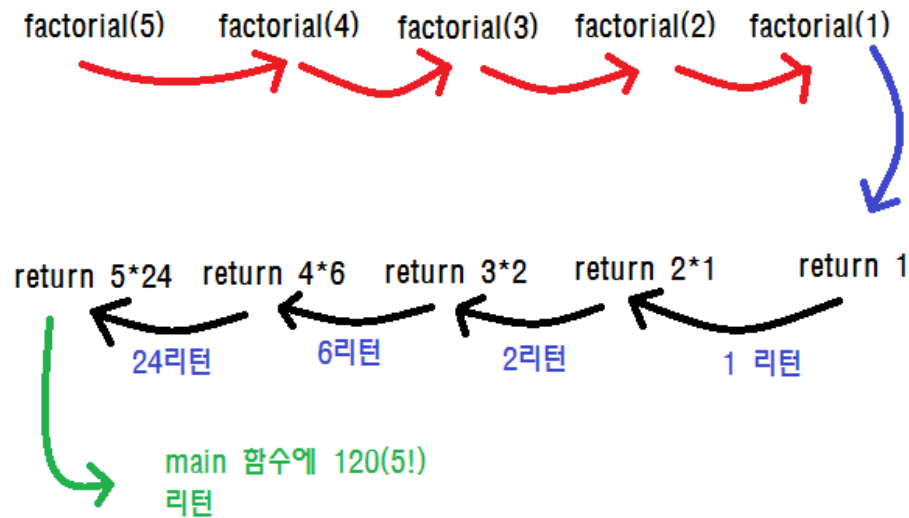
$$F_n := \begin{cases} 1 & \text{if } n = 1; \\ n \times F_{n-1} & \text{if } n > 1. \end{cases}$$

```
#include <stdio.h>

int factorial(int n)
{
    if(n==1)
        return 1;
    else
        return n * factorial(n-1);
}

int main()
{
    printf("%d\n", factorial(5));
    return 0;
}
```

Factorial 코드 분석



```
#include <stdio.h>

int factorial(int n)
{
    if(n==1)
        return 1;
    else
        return n * factorial(n-1);
}

int main()
{
    printf("%d\n", factorial(5));
    return 0;
}
```


재귀호출의 개념이 많이 쓰이는 분야

- 자료구조, 알고리즘
- 예: 아래의 **크기 순서대로 정렬된** 배열에서 4가 저장된 위치(index)를 찾으려면?

1	3	4	6	7	8	10	13	14
---	---	---	---	---	---	----	----	----

예: 배열에서 특정 값의 위치 찾기

1	3	4	6	7	8	10	13	14
---	---	---	---	---	---	----	----	----

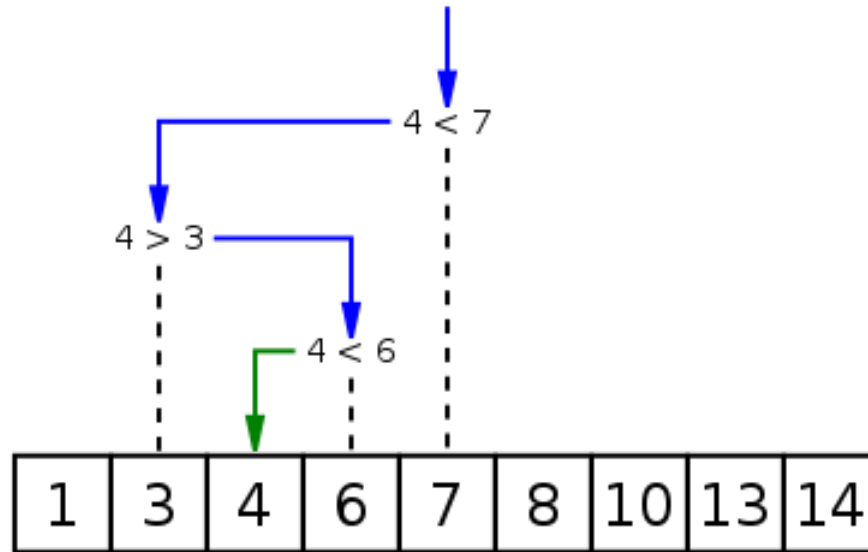
- 그냥 for루프로 처음부터 끝까지 찾아보면 되지 않나요?
- 그런데 길이가 4,294,967,296인 배열이라면?
 - 하필 찾으려는 값이 맨 끝에 위치한다면?

예: 배열에서 특정 값의 위치 찾기

1	3	4	6	7	8	10	13	14
---	---	---	---	---	---	----	----	----

- 더 효율적인 방법?
 - 1) 배열 가운데 위치의 값을 찾고자 하는 값과 비교
 - 1-1) 찾는 값보다 크면 배열의 앞쪽 절반을 대상으로 1)부터 다시 수행
 - -> 1) 과정 중에 또 다시 1) 수행 : 재귀호출
 - 1-2) 찾는 값보다 작으면 배열의 뒤쪽 절반을 대상으로 1)부터 다시 수행
 - -> 1) 과정 중에 또 다시 1) 수행 : 재귀호출
 - 1-3) 찾는 값과 같다면 해당 위치를 찾은 것임
 - -> 재귀함수의 탈출조건

예: 배열에서 특정 값의 위치 찾기



- 이진탐색 (Binary Search)
- 길이 4,294,967,296 배열도 최대 32번만 찾아보면 됨