

학번 : 이름 :

[재귀 함수]

함수와 스택(Stack) 관계 정리하기

- 스택(Stack)의 핵심 원리

스택은 LIFO (Last In First Out) 원칙을 따르며, 주로 Push와 Pop 두 가지 연산으로 동작하는 추상 자료형이에요. 구조상 맨 위의 데이터는 쉽게 꺼낼 수 있지만, 깊숙이 위치한 데이터에 접근하려면 그 위에 쌓인 항목들을 먼저 제거해야 하는 특징이 있어요.

- 함수를 호출 했을 때 스택(Stack) 동작

push, 함수가 호출되면 스택에는 함수의 매개변수, 호출이 끝난 뒤 돌아갈 반환 주소값, 함수에서 선언된 지역 변수 등을 저장해요.

- 함수가 종료 됐을 때 스택(Stack) 동작

pop, 함수가 실행되면서 쌓였던 지역 변수와 매개변수 공간을 날리고 이전 함수의 기본 주소로 되돌아가요.

학번 : 이름 :

- 일반 함수 호출 예시 : 호출 순서와 스택의 변화

```
void b() { printf("b 실행\n"); }
void a() { b(); printf("a 실행\n"); }

int main() {
    a();
    return 0;
}
```

호출 순서와 스택 변화

1단계 : main 호출	2단계 : a()호출	3단계 : b()호출
top bottom main()	top bottom a() main()	top bottom b() a() main()
4단계 : b() 종료	5단계 : a()종료	6단계 : main() 종료
top bottom a() main()	bottom main()	bottom

학번 : 이름 :

- 재귀 함수(Recursive Function)와 스택(Stack)

```
void recFunc(int n) {
    if (n < 1)
        return;      // 종료 조건
    recFunc(n - 1);   // 재귀 호출
}

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

호출 순서와 스택 변화

1단계 : main 호출	2단계 : recFunc(3)호출	3단계 : recFunc(2)호출
top bottom main()	top bottom recFunc(3) main()	top bottom refFunc(2) recFunc(3) main()
4단계 : recFunc(1)호출	5단계 : recFunc(0) 호출	6단계 : recFunc(0) 종료
top bottom refFunc(1) refFunc(2) refFunc(3) main()	top bottom refFunc(0) refFunc(1) refFunc(2) refFunc(3) main()	top bottom refFunc(1) refFunc(2) refFunc(3) main()
5단계 : recFunc(1) 종료	6단계 : recFunc(2) 종료	7단계 : recFunc(3) 종료

학번 : 이름 :

top refFunc(2) refFunc(3) bottom main()	top refFunc(3) bottom main()	bottom main()
8단계 : main 종료	9단계	
bottom main()	bottom	종료 조건이 없으면, 스택에 무한히 쌓여 <u>스택 오버플로우(Stack Overflow)</u>가 발생한다.

- 재귀 함수(Recursive Function)를 쓰는 이유

- ① 가독성과 직관성, 알고리즘 자체가 재귀적인 구조일 때, 이를 수학적 정의 그대로 옮길 수 있어요.
- ② 변수 사용 최소화, 상태를 유지하기 위해 변경 가능한 상태 변수를 명시적으로 선언하고 관리할 필요가 줄어들어, 프로그램의 부수 효과를 방지할 수 있어요.
- ③ 분할, 복잡하고 거대한 문제를 동일한 형태의 작은 하위 문제로 쪼개어 해결하는 구조를 매우 직관적으로 구현할 수 있어요,

- 반복문 vs. 재귀 함수(Recursive Function)

	반복문	재귀함수
코드	<pre>int main() { int n = 3; while (n >= 1) { printf("%d ", n); n = n - 1; } return 0; }</pre>	<pre>void recFunc(int n) { if (n < 1) return; printf("%d ", n); recFunc(n - 1); } int main() { recFunc(3); return 0; }</pre>

학번 : 이름 :

동작원리	명령어가 조건식을 만족하는 동안 메모리의 동일한 코드 블록을 CPU의 제어 흐름 이동을 통해 반복 실행해요.	함수가 자기 자신을 다시 호출하면, 메모리의 스택 영역에 새로운 스택 프레임이 계속 누적되며 진입합니다. 종료 조건을 만나면 역순으로 복귀해요.
메모리 사용	초기 선언된 지역 변수와 제어 변수 메모리만 계속 재사용하므로 메모리 오버헤드가 거의 없어요. ($O(1)$)	함수가 호출될 때마다 매개변수, 지역변수, 복귀 주소를 포함한 스택 프레임이 계속 쌓이므로 메모리 소모가 커요. ($O(N)$)
실행속도	컨텍스트 스위칭이나 스택 관리 비용이 없기 때문에 일반적으로 상대적으로 빠릅니다.	함수 호출 및 복귀 시 발생하는 오버헤드로 인해 상대적으로 느립니다.
위험요소	무한 루프 발생 시 CPU 자원을 과도하게 소비할 수 있으나, 프로세스가 강제 종료되지 않는 한 00M으로 종료되진 않아요.	종료 조건이 잘못되거나 재귀 깊이가 너무 깊어지면 스택 영역을 초과하는 Stack Overflow로 인해 프로그램이 즉시 충돌해요.
코드 가독성	정적이고 직관적이어서 흐름을 추적하기 쉬우나, 트리 순회, DFS, 분할 정복 같은 알고리즘을 구현하면 코드가 비대해지고 가독성이 떨어질 수 있어요.	수학적 정의나 도메인 논리를 그대로 코드로 옮길 수 있어 복잡한 알고리즘에서 코드가 매우 간결하고 직관적입니다. 다만, 생각의 흐름이 비선형적이라 디버깅이 까다로울 수 있습니다.

1. 실습예제1-recEx01.c

- 코드와 출력 결과 붙여 넣기

코드	출력결과
<pre>#include <stdio.h> void recFunc(int n) { if (n == 0) return; printf("%d ", n); recFunc(n - 1); } int main() { recFunc(9); return 0; }</pre>	<pre>9 8 7 6 5 4 3 2 1</pre>

학번 : 이름 :

2. 실습예제2-recEx02.c

- 코드와 출력 결과 붙여 넣기

코드	출력결과
<pre>#include <stdio.h> int recOddSum(int n) { if (n <= 0) return 0; if (n % 2 == 0) return recOddSum(n - 1); return n + recOddSum(n - 2); } int main() { int n; printf("n = "); if (scanf("%d", &n) != 1) { return 1; } printf("합=%d\n", recOddSum(n)); return 0; }</pre>	<pre>n = 10 합=25</pre>

3. 실습예제3-recEx03.c

- 코드와 출력 결과 붙여 넣기

코드	출력결과
<pre>#include <stdio.h> int recSum(int n) { if (n == 0) return 0; return n + recSum(n - 1); } int main() {</pre>	<pre>합=55</pre>

학번 : 이름 :

<pre>printf("합=%d\n", recSum(10)); return 0; }</pre>	
--	--

4. 실습예제4-recEx04.c

- 코드와 출력 결과 붙여 넣기

코드	출력결과
<pre>#include <stdio.h> int recPower(int a, int b) { if (b == 0) return 1; return a * recPower(a, b - 1); } int main() { int a, b; printf("a, b = "); scanf("%d %d", &a, &b); printf("%d^%d=%d\n", a, b, recPower(a, b)); return 0; }</pre>	<pre>a, b = 10 2 10^2=100</pre>

5. 실습예제5-recEx05.c

- 코드와 출력 결과 붙여 넣기

코드	출력결과
<pre>#include <stdio.h> int recSum(int from, int to) { if (from > to) return 0; return from + recSum(from + 1, to); } int main() { int from, to; printf("from, to = "); if (scanf("%d %d", &from, &to) != 2) { return 1; } }</pre>	<pre>from, to = 3 5 합=12</pre>

학번 : 이름 :

<pre>printf("합=%d\n", recSum(from, to)); return 0; }</pre>	
--	--

학번 : 이름 :

[라이브러리(내장) 함수]

1. 실습예제1-lib_func_exam01.c

- 코드와 출력 결과 붙여 넣기

코드	출력결과

2. 실습예제2-lib_func_exam02.c

- 코드와 출력 결과 붙여 넣기

코드	출력결과