순수 함수 vs Side-effect 함수

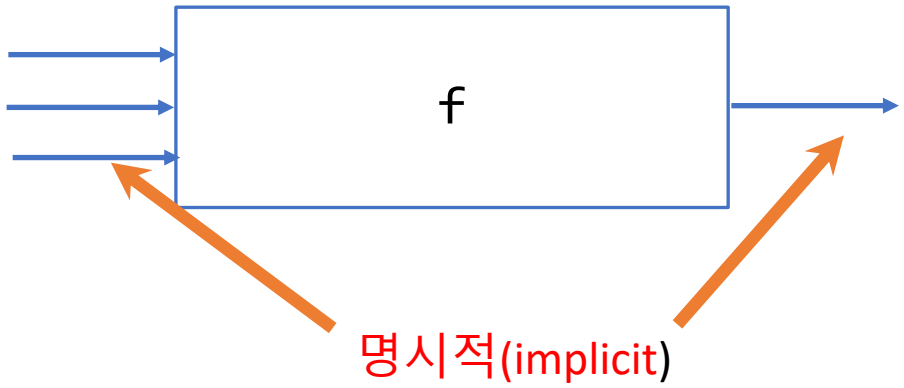
- 순수 함수
  - 계산에 필요한 모든 정보를 인수로 표현

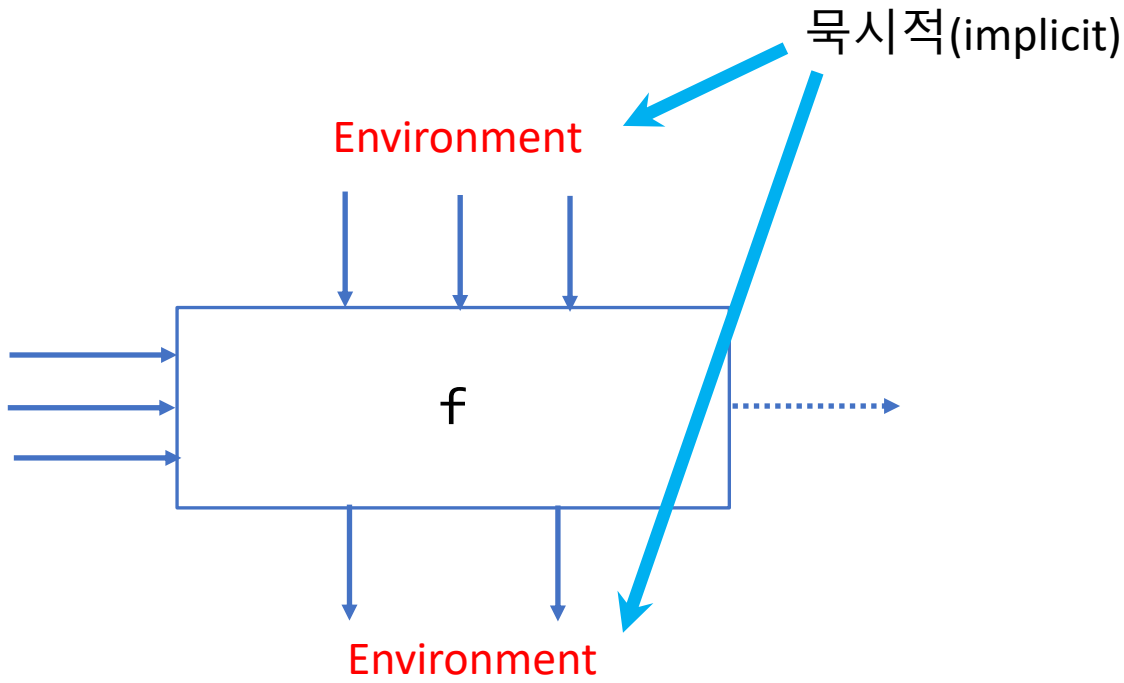
```
#include <stdio.h>

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

int main () {
  int n = 4;
  printf ("fac(%d) is %d\n", n, fac(n));
}
```

- Side-effect 함수
  - 인수 외에 Environment가 개임

```
#include <stdio.h>

int result = 1; // global

int fac (int n) {
  for (int i = n; i != 1; i--)
    result = result * i;
}

int main () {
  int n = 4;
  printf ("fac(%d) is %d\n", n, result);
}
```

순수 및 Side-effect 함수





- 순수 함수
  - 오직 입력 인수에 의해서만 영향을 받음
  - 그 이외의 영향을 받지 않음
  - 입력 인수와 return 값이 반드시 존재함
  - Return 값은 유일함 (unique)
  - 함수 실행 후 입력 변수의 값 불변

- Side-effect 함수
  - 인수 외에 묵시적으로 표현되는 environment (전역 변수 등) 개입
  - 입력 인수나 Return 값이 없을 수도 있음
  - Return 값이 유일하지 않을 수 있음
  - 함수 실행 후 입력 변수 값 변화 가능성 있음

순수 함수 이해하기 (reasoning)

- 수식의 값(value)이 유일 함
  - 치환(substitution)에 의한 reasoning (수학처럼)
  - 함수의 계산이 오직 함수 내에서만 처리되며,
  - 문맥에 영향을 받지 않고, 언제나 유일한 값을 제공함
  - 값 위주 (value-oriented)의 reasoning

$$fac(4) = 4 * fac(3) = 4 * 3 * fac(2) = 4 * 3 * 2 * fac(1) = 4 * 3 * 2 * 1 = 24$$

- 참조 투명성 (referential transparency)
- 프로그램을 함수 단위로 modular 하게, 이해/유지/관리할 수 있음
- 대규모 프로그램 구성에 매우 유리함
- 전역 변수 및 reference 객체 (pointer) 사용 금지

순수 함수와 Effect 함수 사용 병행

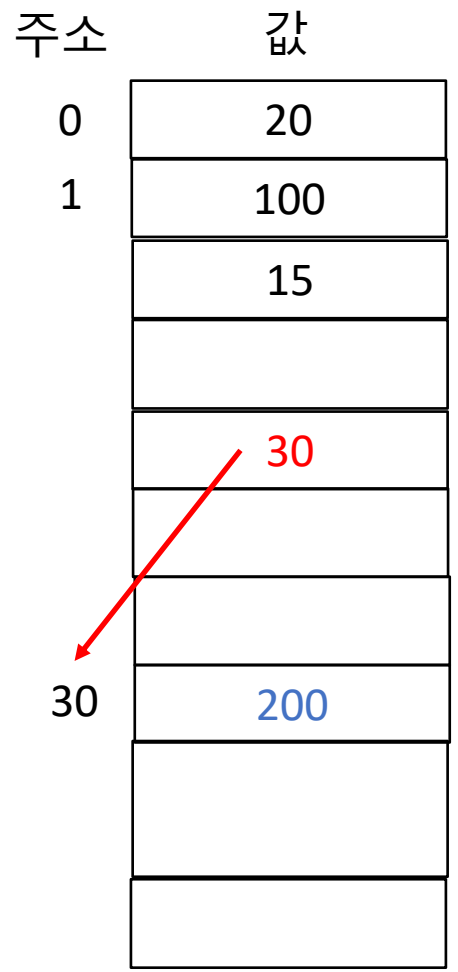
- 가급적 프로그램을 순수 함수 형태로 구성하는 것을 추천함.
- 그러나 모든 계산을 순수 함수 형태로 표현하기는 어려움
- 대표적으로, 입출력(Input/Output) 및 Event 프로그래밍은 effect를 사용하지 않을 경우, 코드가 매우 복잡해 지므로 순수 함수형 프로그래밍은 비현실적

- Haskell
  - 순수 함수형 프로그래밍
  - 입출력에 대해서는 모나드(Monad)를 이용하는 effect 프로그래밍 적용
  - 타입에 의해서 순수/비순수 함수가 구분함으로써, 순수성을 유지함

von Neumann 구조

- Random Access Memory 구조
  - 메모리는 Byte 단위로 주소(자연수)를 가짐
  - CPU가 메모리를 접근할 때는 주소를 이용함
  - 메모리 주소에 상관없이 일정한 시간에 접근함
  - CPU가 처리하는 모든 데이터는 메모리에 적재(load) 되어야 함.
    CPU는 하드디스크나 다른 장치의 데이터를 직접 접근하지 않음.
  - Turing Machine과 유사한 구조

- Reference 값
  - 메모리 내에 값(value)이 아닌 주소(address) 를 저장하는 형태의 프로그래밍 기법이 가능함
  - 에셈블리의 indirect addressing
  - C의 pointer, array
  - Java의 reference 타입 변수 (Objects, arrays)



명령형(Imperative) 프로그래밍의 변수

- Turing Machine과 Assembly 프로그래밍: 메모리 중심(Memory-oriented) 의 계산
  - 메모리에 저장된 값을 읽어, 연산 후, 그 결과를 다시 메모리에 저장함

- 명령형 프로그램의 변수
  - 메모리 한 cell을 추상화하여, 이름을 부여함
  - 메모리의 read 및 write가 가능함
  - Reference 타입의 변수는 값으로서 메모리 주소를 저장함 (pointer)
  - 변수의 L-value, R-value

    `x = x + 1` (x의 값을 읽어, 1과 더한 결과 값을, x의 주소에 저장함)

명령형 언어의 구성

- 프로그램은 함수들로 구성됨
  - 식(expression)
    - 산술식, 논리식
- 한 함수는 대략 다음과 같은 문장들의 연속
  1. 할당문
  2. If-문
  3. While 문 (반복)
  4. 복합문 { }
  5. 문장의 연속 ; 으로 표현됨

```
int fac(int n) {
  int x = 1;
  while (n > 1) {
    x = x * n;
    n = n - 1;
  }
  return x;
}
```

명령형 언어의 실행 순서

- 프로그래머가 모든 실행 순서를 명시적으로 표현함
  - 하드웨어 구조에 좀 더 가까운 형태

- 위의 문장에서 아래 문장으로
- 수식에서는 왼쪽에서 오른쪽으로

- 예

```
(1+2) * (3+4)
f(10*20, 1+2)
```

- 순수 함수형 언어
  - 선언적(declarative)
  - 실행 순서를 표현하지 않는다 (실행 순서는 시스템이 결정함)

메모리에 대한 추상화

- 배열(array)
  - RAM의 특성을 활용하는 프로그래밍 방법
  - 예: int a[1000000] 로 선언할 때

    a[i] 의 주소 = a의 시작 주소 + i*4
  - 배열 크기에 상관없이 접근이 상수 $O(c)$

- 수학의 변수와 명령형 언어 변수의 차이
  - 수학에서는 변수를 메모리와 연관시키지 않지만, 명령형 언어에서는 메모리에 사이즈를 결부시킴 (예. 정수는 4byte)
  - 수학에서는 함수의 인수가 주어지면 정의 부분에서 이 값이 변함이 없지만, 명령형 언어에서는 할당문에 의해 이 값이 동적으로 변화될 수 있음

```
f(int x) {

   x = x + 1;

   x = y + z;

}
```

순수 함수형(Haskell) 언어 문법의 특징

• 할당문 사용하지 않음	=> 상수 (한 번만 할당)
• 변수의 동적 변환 사용	=> 변수의 동적 변화 없음
• 변수의 타입과 메모리 바이트 수	=> 변수를 메모리와 연관시키지 않음
• Mutable 변수 및 List	=> Immutable 자료구조
• 배열의 In-line 업데이트	=> 새로운 자료 구조 복사
• 문장(statement)	=> 식 (expression). 문장은 존재 안함
• 문장의 연속 ;	=> 함수의 합성 (함수 호출)
• 조건문	=> conditional expression
• 반복 (while, for)	=> 재귀 함수

객체지향 프로그래밍에서 변수의 Scope

- 전역 변수의 사용은 제한되어야 함

- 전역 변수 사용의 편리성을 유지하면서, 클래스 내에 정의된 메소드들 사이에 변수들을 전역적으로 사용