# 임베디드 소프트웨어

# < 데이터 구조 >

## 1, 데이터 구조의 이해

### 1) 알고리즘의 표현과 분석

▶ 알고리즘은 어떤 문제를 해결하기 위한 명확한 절차나 방법을 의미한다. 알고리즘의 표현과 분석은 컴퓨터 과학에서 중요한 주제로, 효율적인 프로그램을 작성하고 최적의 솔루션을 찾는 데 필수적이다.

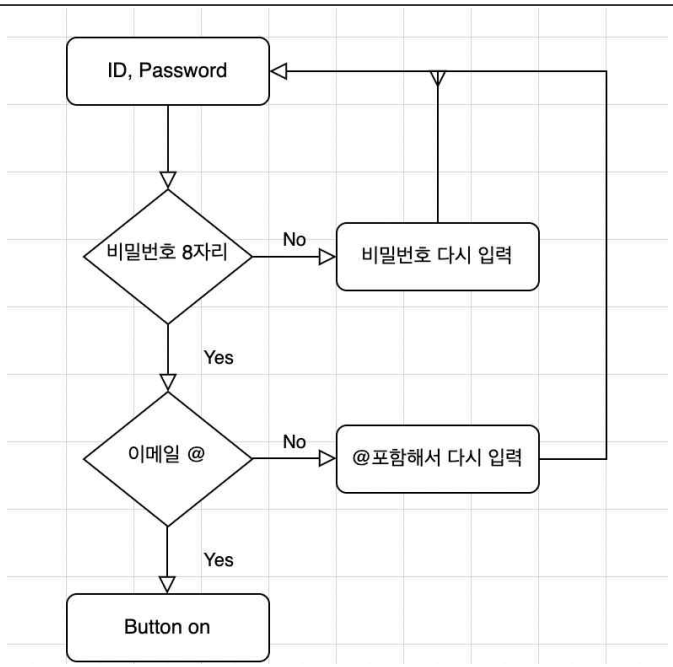
▶ 알고리즘의 표현: 알고리즘이 어떻게 기술되는지에 대한 내용이다. 이는 일반적으로 두 가지 방식으로 이루어진다.

(1) 의사 코드(Pseudocode): 이는 프로그래밍 언어와 유사하지만, 보다 간단하고 일반적인 형태로 알고리즘을 기술하는 방법이다. 의사 코드는 특정 프로그래밍 언어에 종속되지 않으므로, 알고리즘이 수행하는 작업에 집중할 수 있다.

```
stand up
assign yourself the number 1
until only one person remains standing
    pair off with someone else standing
    add your numbers together
    assign yourself the new number
    choose one member of the pair to sit
    if you are chosen
        sit down and do nothing else
```

< 의사 코드 예시 >

(2) 플로우차트(Flowchart): 이는 그림이나 도형을 사용하여 알고리즘이 어떻게 작동하는지 시각적으로 나타내는 방법이다. 각 단계와 결정점이 독특한 모양으로 표시되며, 화살표를 사용하여 순서와 흐름을 나타낸다.




< 플로우 차트 예시 >

▶ 알고리즘 분석: 알고리즘이 얼마나 "좋은" 지를 평가하는 과정이다. 주요 관심사항은 시간 복잡도(알고리즘이 실행되는 데 필요한 시간)와 공간 복잡도(알고리즘이 실행되기 위해 필요한 메모리)이다.

(1) 시간 복잡도(Time Complexity): 입력 크기에 대해 계산량이 어떻게 증가하는지를 설명한다. O-표기법(Big O notation)은 일반적으로 가장 많이 사용되며, 최악의 경우 성능을 나타내주므로 보수적인 측면에서 유용하다.

(2) 공간 복잡도(Space Complexity): 입력 크기에 대해 메모리 사용량이 어떻게 증가하는지를 설명한다. 알고리즘이 많은 양의 데이터를 저장해야 한다면, 이는 중요한 고려 사항이 될 수 있다.

**2) 배열**

▶ 배열(array)은 컴퓨터 과학에서 가장 기본적인 데이터 구조 중 하나이다. 배열은 동일한 타입의 여러 개의 요소를 연속적인 메모리 공간에 저장하는 구조로, 각 요소는 인덱스(index)로 참조할 수 있다.

▶ 특징

(1) 고정된 크기: 배열은 생성 시점에 그 크기가 결정되며, 일단 생성하면 크기를 변경할 수 없다. 이는 동적으로 크기가 변하는 데이터 구조와는 대비된다.

(2) 인덱스 접근: 배열의 각 요소는 인덱스를 사용하여 시간 내에 접근할 수 있다. 즉, 어떤 위치든 상관없이 해당 위치의 요소를 빠르게 읽거나 쓸 수 있다.

(3) 연속적인 메모리 할당: 배열의 모든 요소는 연속된 메모리 주소에 저장된다. 이

로 인해 CPU 캐시 효율성이 향상되어 성능이 개선될 수 있다.

▶ 장점

(1) 빠른 접근 속도: 인덱스를 사용한 직접 접근으로 인해, 배열은 임의 위치에 있는 데이터에 매우 빠르게 접근할 수 있다는 장점이 있다.

(2) 간단함: 구현이 간단하고 사용하기 쉽다는 장점이 있다.

▶ 단점

(1) 크기 변경 불가능: 한 번 생성된 배열은 크기를 변경할 수 없으므로, 추가적인 공간이 필요하거나 공간을 줄여야 할 때 유연성이 부족하다.

(2) 공간 낭비 가능성: 배열을 선언할 때 모든 공간을 미리 할당하므로 실제로 모든 공간을 사용하지 않아도 메모리가 낭비될 가능성이 있다.

(3) 삽입 및 삭제 비효율성: 중간 위치에서 원소를 삽입하거나 삭제하는 경우, 다른 원소들을 이동시켜야 하므로 비효율적이다.

▶ 배열은 자료형, 변수 이름, 배열의 크기, 변수의 값을 정해줌으로서 사용 가능하며, 배열의 인덱스는 0으로부터 시작한다..

```
int array[4] = {1,2,3,4}
array[0] = 1, array[1] = 2, array[2] = 3, array[3] = 4
                          < 1차원 배열의 사용 예시 >
int arr[2][3] = {{1,2,3},{4,5,6}}
arr[0][0] = 1, arr[0][1] = 2, arr[0][2] = 3
arr[1][0] = 4, arr[1][1] = 5, arr[1][2] = 6
                          < 2차원 배열의 사용 예시 >
```

### 3) 연결 리스트

▶ 연결 리스트(linked list)는 컴퓨터 과학에서 중요한 데이터 구조 중 하나이다. 연결 리스트는 노드(node)라는 요소들이 포인터를 통해 서로 연결되어 있는 구조이다. 각 노드는 데이터를 담고 있는 요소와 다음 노드를 가리키는 포인터로 구성된다.

▶ ListNode 정의

```
// 노드 구조체 정의
typedef struct Node {
    int data;
    struct Node* next;
} Node;
```

▶ 노드 생성 함수

```
// 노드 생성 함수
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
```

▶ 노드 추가 함수

```
// 노드 추가 함수
void addNode(Node** head, int data) {
    if (*head == NULL) {
        *head = createNode(data);
    } else {
        Node* tmp = *head;
        while (tmp->next != NULL) {
            tmp = tmp->next;
        }
        tmp->next = createNode(data);
    }
}
```

▶ 연결리스트 출력 함수

```
// 연결 리스트 출력 함수
void printList(Node* head) {
    Node* tmp = head;
    while (tmp != NULL) {
        printf("%d ", tmp->data);
        tmp = tmp->next;
    }
    printf("\n");
}
```

▶ 메인 함수

```
// 메인 함수
int main() {
    Node* head = NULL;
    addNode(&head, 1);
    addNode(&head, 2);
    addNode(&head, 3);
    printList(head);
    return 0;
}
```

: 이 코드는 연결 리스트를 생성하고 노드를 추가하며, 연결 리스트를 출력하는 기본적인 동작을 수행한다. 출력 결과는 '1 2 3'이 된다.

▶ 특징

(1) 동적 크기: 연결 리스트의 크기는 동적으로 변할 수 있다. 즉, 실행 시간 동안 노드를 추가하거나 삭제함으로써 리스트의 크기를 변경할 수 있다.

(2) 논리적 순서: 연결 리스트의 노드들은 메모리 내에서 물리적으로 분산되어 있을 수 있지만, 각 노드가 다음 노드를 가리키는 포인터에 의해 로직적인 순서가 유지된다.

▶ 장점

(1) 동적 크기: 필요에 따라 동적으로 원소를 추가하거나 삭제할 수 있다는 점이 배열과 비교했을 때 주요한 장점이다.

(2) 삽입 및 삭제 효율성: 연결 리스트에서 원소의 삽입 및 삭제 작업은 시간 복잡도로 이루어질 수 있다(단, 해당 위치에 대한 참조가 이미 있다고 가정). 배열과 비교했을 때 중간 위치에 대한 삽입 및 삭제 작업이 훨씬 효율적이다.

▶ 단점

(1) 랜덤 액세스 비효율성: 인덱스로 바로 접근하는 것이 아니라 첫 번째 노드부터 찾아야 하므로, 임의 위치에 접근하는 것은 O(n) 시간 복잡도가 소요된다.

(2) 추가 메모리 사용: 각각의 요소(노드)마다 데이터 외에도 포인터 정보를 저장해야 하므로 추가 메모리 공간이 필요하다.

(3) 캐시 활용성 부족: 연속된 메모리 공간을 사용하지 않으므로 CPU 캐시의 효율성이 떨어질 수 있다.

**4) 스택과 큐**

▶ 스택(stack)과 큐(queue)는 컴퓨터 과학에서 중요한 추상 데이터 타입이다. 두 데이터 구조 모두 일련의 요소를 저장하지만, 요소에 접근하고 조작하는 방식이 다

르다.

▶ 스택(Stack)

: 스택은 LIFO(Last In, First Out) 정책을 따르는 선형 데이터 구조이다. 즉, 가장 최근에 스택에 추가된 항목이 가장 먼저 제거된다. 스택은 다음과 같은 주요 연산을 지원합니다.

(1) Push: 스택의 맨 위에 항목을 추가한다.

(2) Pop: 스택의 맨 위에서 항목을 제거하고 반환한다.

(3) Peek/Top: 스택의 맨 위의 항목을 조회한다(제거하지 않음).

(4) IsEmpty: 스택이 비어 있는지 확인한다.

▶ 스택은 함수 호출/반환, 후위 표기법 계산, 깊이 우선 탐색(DFS), 백트래킹 등 다양한 알고리즘과 문제 해결 전략에서 사용된다.

▶ 큐(Queue)

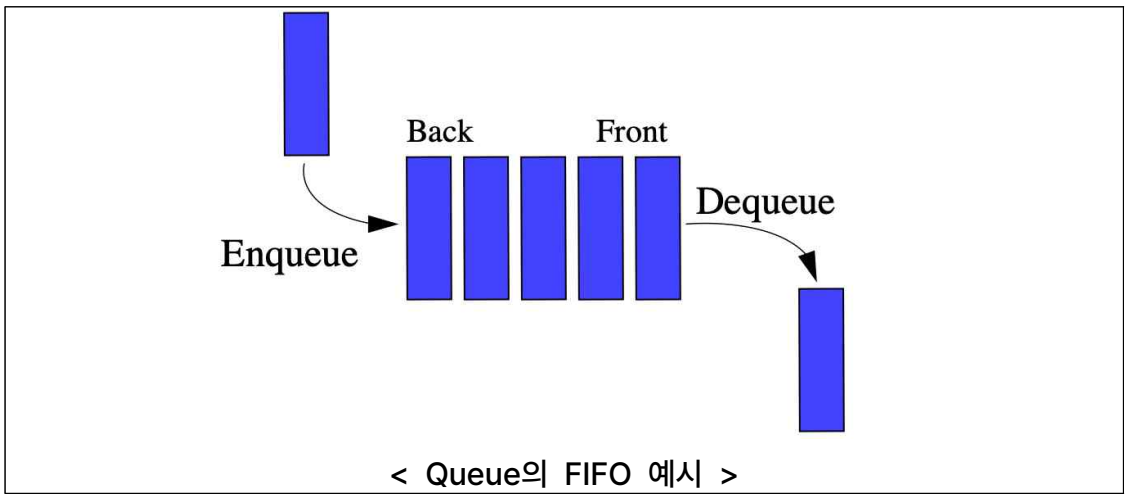
: 큐는 FIFO(First In, First Out) 정책을 따르는 데이터 구조이다. 즉, 가장 먼저 큐에 들어간 항목이 가장 먼저 나온다. 큐는 다음과 같은 주요 연산을 지원한다.

(1) Enqueue: 큐의 끝(rear)에 항목을 추가한다.

(2) Dequeue: 큐의 시작(front)에서 항목을 제거하고 반환한다.

(3) Front/Peek: 큐의 시작 위치(front)의 항목을 조회한다(제거하지 않음).

(4) IsEmpty: 큐가 비어 있는지 확인한다.




< Queue의 FIFO 예시 >

▶ 큐는 너비 우선 탐색(BFS), CPU 스케줄링, 캐시 구현 등 여러 알고리즘과 시스템에서 사용된다.

▶ 큐의 핵심 연산

(1) push(item) : 큐 맨 뒤에 데이터 삽입

(2) pop() : 큐 맨 앞 데이터 삭제 및 반환

(3) front() : 큐 맨 앞 데이터 반환

(4) back() : 큐 맨 뒤 데이터 반환

(5) size() : 큐 크기 반환

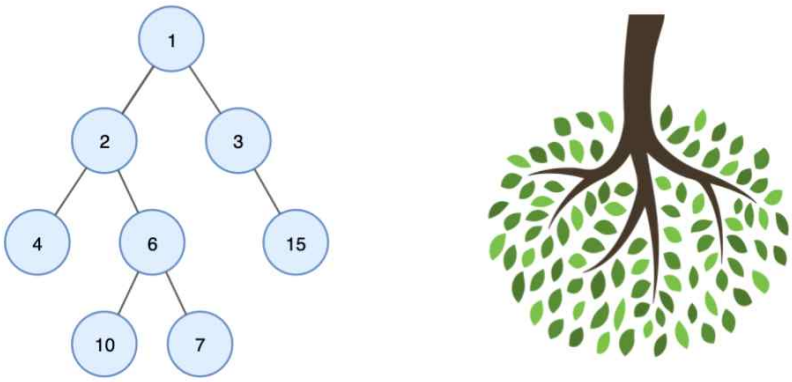
(6) empty() : 큐가 비었는지 여부 반환

(7) full() : 큐가 가득 찼는지 여부 반환

### 5) 트리

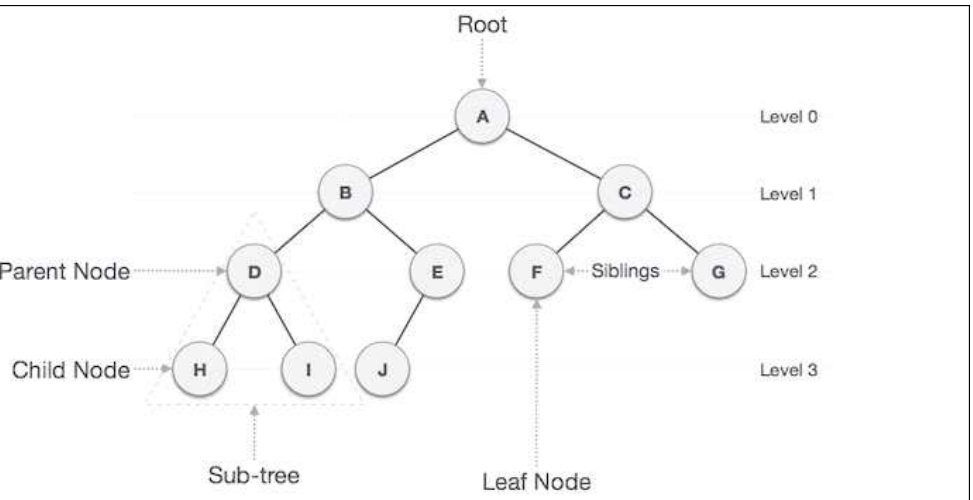
▶ 트리(Tree)

: 트리는 비선형 데이터 구조로, 요소들이 계층적 관계를 가지는 방식으로 구성된다. 트리는 노드(node)라고 불리는 요소들로 이루어져 있으며, 각 노드는 다른 노드에 대한 연결(일반적으로 '링크'나 '포인터'라고 부름)을 가진다.




< 트리 구조 예시 >

▶ 주요 개념



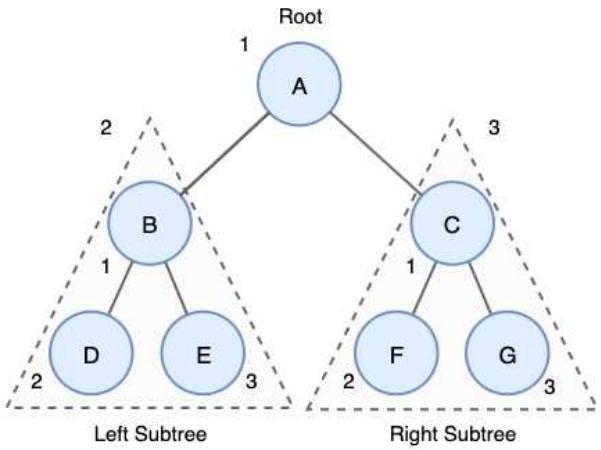

< 트리 구조 예시 >

(1) Node(노드): 트리의 각 요소를 노드라고 한다. 노드는 데이터를 저장하며, 다른 노드와의 연결 정보를 가지고 있다.(ex. A, B, C, D, E, F, G, H, I, J)

(2) Edge(간선): 노드와 노드 사이를 연결하는 선을 간선이라고 한다.
(3) Root(루트): 트리의 시작점인 노드를 루트라고 한다. 트리에는 하나의 루트만 존재한다.(A)
(4) Parent(부모): 어떤 노드의 직전 단계에 연결된 노드를 그 노드의 부모라고 한다.(ex. H, I의 부모 노드는 D)
(5) Child(자식): 어떤 노드의 직후 단계에 연결된 노드를 그 노드의 자식이라고 한다.(ex. 노드 D의 자식 노드는 H, I)
(6) Leaf(리프): 자식이 없는 노드를 리프 노드(Leaf Node) 또는 단말 노드(Terminal Node) 또는 외부 노드(External Node)라고 한다.(ex. H, I, J, F, G)
(7) branch(가지): 자식을 하나 이상 가지는 노드이며 가지 노드(Branch Node) 또는 비 단말 노드(Non-terminal Node) 또는 내부 노드(Internal Node)라고 한다.(ex. A, B, C, D, E)
(8) Sibling(형제): 같은 부모를 가진 노드들을 형제 노드라고 한다.(ex. H, I는 같은 부모를 가지는 형제 노드)
(9) Depth(깊이): 루트에서 특정 노드에 이르기까지의 간선의 수를 그 노드의 깊이라고 한다.(ex. 루트 노드의 깊이 : 0, D의 깊이 : 2)
(10) Height(높이): 트리의 높이는 루트 노드에서 가장 깊숙히 있는 노드까지의 깊이이다.(ex. 리프 노드의 높이 : 0, A노드의 높이 : 3)
(11) Degree(차수): 트리의 노드 중에서 가장 많은 자식을 가진 노드의 자식의 수를 말한다. 즉, 어떤 트리에서 각 노드의 차수를 확인하고, 그 중에서 가장 큰 값을 찾는 것이 트리의 차수를 계산하는 방법이다.

▶ 트리의 순회(Traversal)

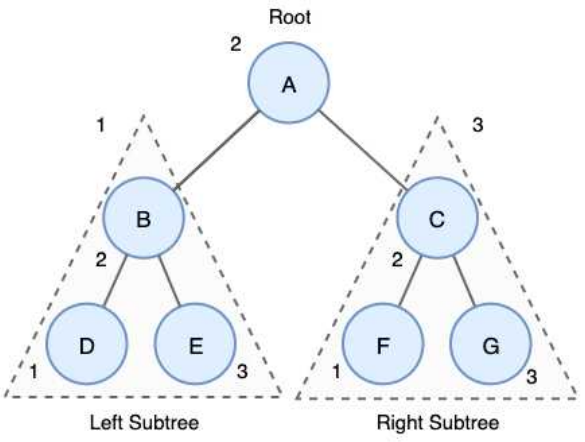
: 트리의 모든 노드를 체계적으로 방문하는 것을 말한다. 트리의 순회 방법에는 전위 순회(Preorder), 중위 순회(Inorder), 후위 순회(Postorder) 등이 있다.
(1) 전위 순회(Preorder): 트리의 노드를 '부모 노드 -> 왼쪽 자식 -> 오른쪽 자식' 순서로 방문하는 방법이다. 따라서 부모 노드가 자식 노드보다 먼저 처리된다. 이 방법은 트리의 복사나 트리의 표현식 계산 등에 사용된다.




< 전위 순회 예시 >

: A→B→D→E→C→F→G

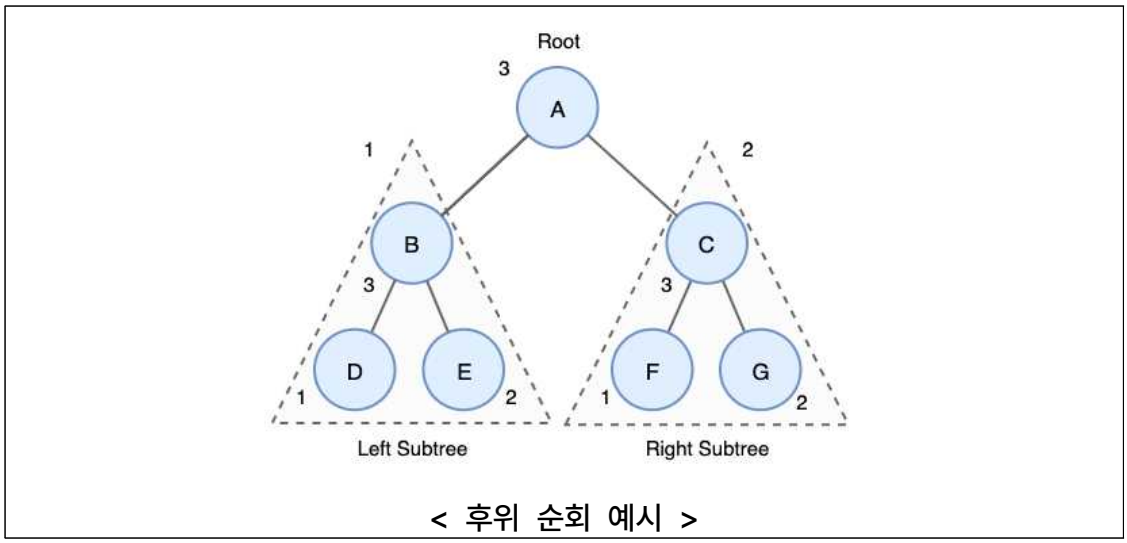
(2) 중위 순회(Inorder): 트리의 노드를 '왼쪽 자식 -> 부모 노드 -> 오른쪽 자식' 순서로 방문하는 방법이다. 이 방법은 이진 탐색 트리에서 사용하면 키 값을 오름차순으로 얻을 수 있어 정렬에 사용된다.




< 중위 순회 예시 >

: D→B→E→A→F→C→G

(3) 후위 순회(Postorder): 트리의 노드를 '왼쪽 자식 -> 오른쪽 자식 -> 부모 노드' 순서로 방문하는 방법이다. 이 방법은 트리의 노드를 처리하기 전에 그 노드의 모든 자식 노드를 먼저 처리해야 하는 경우에 유용하다.




< 후위 순회 예시 >

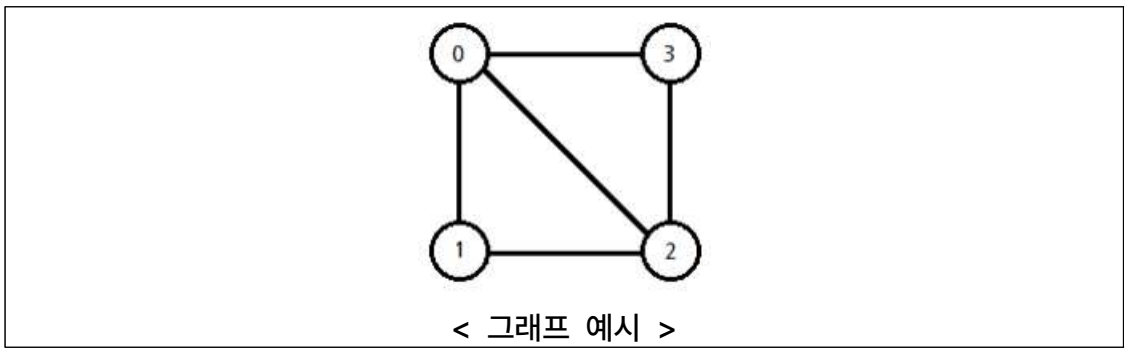
: D→E→B→F→G→C→A

### 6) 그래프

▶ 그래프(Graph)

: 그래프도 비선형 데이터 구조지만, 그래프 내에서 어떤 요소든 다른 어떤 요소와도 직접적으로 연결될 수 있다. 그래프에서 요소들은 일반적으로 '노드' 혹은 '정점(vertex)'라고 부르며, 이러한 정점 사이의 연결을 '간선(edge)'라고 한다.
그래프는 트리와는 다르게 각 노드마다 간선이 있을 수도 있고 없을 수도 있으며, 루트노드와 부모와 자식이라는 개념이 존재하지 않는다.

▶ 그래프의 주요 개념




< 그래프 예시 >

(1) 정점(Vertex) : 노드(node) 라고도 하며 정점에는 데이터가 저장된다. (0, 1, 2, 3)

(2) 간선(Edge) : 정점(노드)를 연결하는 선으로 link, branch 라고도 부른다.

(3) 인접 정점(adjacent Vertex) : 간선에 의해 직접 연결된 정점이다. (0과 2은 인접정점)

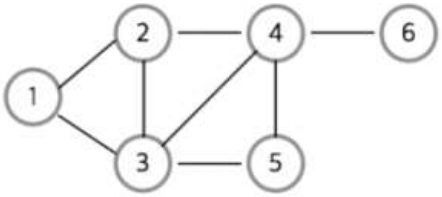
(4) 단순 경로(simple path) : 경로 중에서 반복되는 정점이 없는 경우에 한붓그

리기와 같이 같은 간선을 지나가지 않는 경로이다. ( 0->3->2->1 은 단순경로 )
(5) 차수(degree) : 무방향 그래프에서 하나의 정점에 인접한 정점의 수이다. (0의 차수는 3)
(6) 진출 차수(in-degree) : 방향 그래프에서 외부로 향하는 간선의 수이다.
(7) 진입 차수(out-degree) : 방향 그래프에서 외부에서 들어오는 간선의 수이다.
(8) 경로 길이(path length) : 경로를 구성하는데 사용된 간선의 수이다.
(9) 사이클(cycle) : 단순 경로의 시작 정점과 종료 정점이 동일한 경우이다.

▶ 그래프의 구현 방법

: 그래프를 구현하는 방법에는 '인접행렬'과 '인접리스트'방식이 있다. 두 개의 구현 방식은 각각의 상반된 장단점을 가지고 있다.

(1) 인접행렬 방식



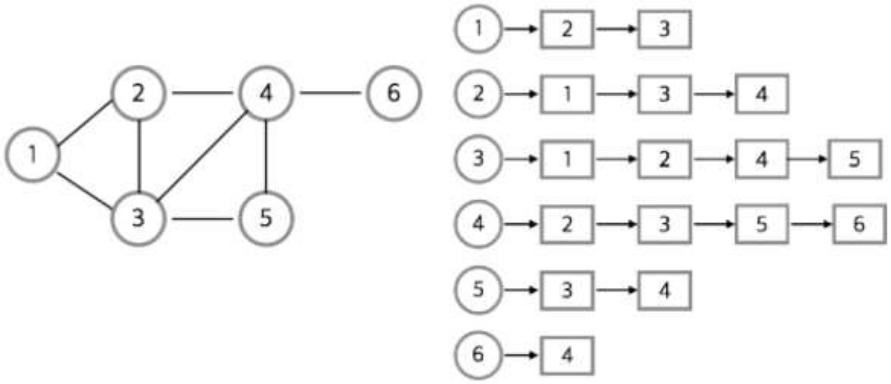

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 1 | 0 | 0 |
| 3 | 1 | 1 | 0 | 1 | 1 | 0 |
| 4 | 0 | 1 | 1 | 0 | 1 | 1 |
| 5 | 0 | 0 | 1 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 1 | 0 | 0 |

< 인접행렬 방식 예시 >

: 인접행렬은 그래프의 노드를 2차원 배열로 만든 것이다. 노드들 간에 직접 연결이 되어있으면 1을, 아니면 0을 넣어서 행렬을 완성시킨다.

- 인접행렬의 장점
  · 2차원 배열 안에 모든 정점들의 간선 정보가 담겨있기 때문에 두 정점에 대한 연결 정보를 조회할 때 O(1) 의 시간복잡도면 가능하다.
  · 인접리스트에 비해 구현이 쉽다.
- 인접행렬의 단점
  · 모든 정점에 대해 간선 정보를 대입해야 하므로 $O(n^2)$의 시간복잡도가 소요된다.
  · 무조건 2차원 배열이 필요하기 때문에 필요 이상의 공간이 낭비된다.

(2) 인접리스트 방식




< 인접리스트 방식 예시 >

: 인접리스트는 그래프의 노드를 리스트로 표현한 것이다. 주로 정점의 리스트 배열을 만들어 관계를 설정하며 노드들 간에 직접 연결이 되어있으면 해당 노드의 인덱스에 그 노드를 삽입해주면 된다. 즉, 1에는 2와 3이직접 연결되어 있기 때문에 배열의 1번째 칸에 2와 3을 넣어준다.

- 인접리스트의 장점

· 정점들의 연결 정보를 탐색할 때 O(n) 시간이면 가능하다.

· 필요한 만큼의 공간만 사용하기 때문에 공간의 낭비가 적다.

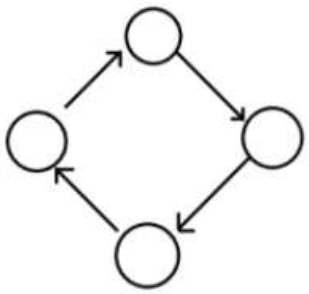
- 인접리스트의 단점

· 특정 두 점이 연결되었는지 확인하려면 인접행렬에 비해 시간이 오래걸린다.
(O(E) 시간 소요. E는 간선의 개수)

· 구현이 비교적 어렵다.

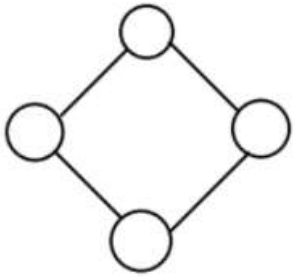
▶ 그래프의 주요 종류

(1) 방향 그래프(Directed Graph): 모든 간선이 방향을 가지는 그래프이며, 간선의 방향으로만 이동할 수 있다.



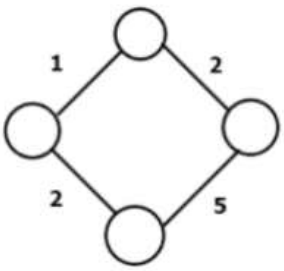
< 방향 그래프 예시 >

(2) 무방향 그래프(Undirected Graph): 간선에 방향이 없어서 양방향으로 이동할 수 있는 그래프이다.



< 무방향 그래프 예시 >

(3) 가중치 그래프(Weighted Graph): 간선에 값(가중치)가 할당된 그래프로서 두 정점을 이동할 때 비용이 드는 그래프이다.




< 가중치 그래프 예시 >

# < 프로그래밍 >

## 1. C프로그래밍

### 1) 데이터 타입과 연산자

▶ C 언어는 다양한 종류의 데이터 타입을 제공한다. 이들은 크게 기본 데이터 타입, 열거형, 구조체, 공용체 등으로 나눌 수 있다.

▶ 기본 데이터 타입

: C 언어에서 가장 기본적인 형태의 데이터를 저장하는 데 사용된다.

(1) int: 정수를 표현하는데 사용된다. 메모리에서 차지하는 크기는 플랫폼에 따라 다르지만, 일반적으로 4바이트를 사용한다.

(2) float: 실수를 표현하는데 사용된다. 메모리에서 차지하는 크기는 일반적으로 4바이트이다.(ex. float num = 3.14f;) 소수점 이하의 자릿수를 제한하려면 %.2f와 같이 형식 지정자를 사용할 수 있다.

(3) double: 실수를 표현하되, float보다 더 큰 범위와 더 높은 정밀도를 가진다. 메모리에서 차지하는 크기는 일반적으로 8바이트이다.(ex. double num = 3.1415926535897932 38;)

(4) char: 단일 문자를 표현하는데 사용된다. 메모리에서 차지하는 크기는 1바이트이다.

(5) short: "short"는 "int"보다 작은 메모리 공간을 차지하는 정수를 저장하는 데 사용된다. 일반적으로 2바이트를 차지하며, 표현할 수 있는 값의 범위는 -32768에서 32767까지이다. "short int" 또는 간단히 "short"로 선언할 수 있다.

(6) long: "long"은 "int"보다 큰 메모리 공간을 차지하는 정수를 저장하는 데 사용된다. 일반적으로 4바이트를 차지하며, 표현할 수 있는 값의 범위는 약 -2,147,483,648에서 2,147,483,647까지이다. "long int" 또는 간단히 "long"으로 선언할 수 있다.

▶ 출력 형식 지정자

(1) int: %d 또는 %i를 사용한다.

(2) char: %c를 사용한다.

(3) string: %s를 사용한다.

(4) float: %f를 사용한다.

(5) double: %lf를 사용한다.

(6) 10진수: %d 또는 %i를 사용한다.

(7) 8진수: %o를 사용한다.(값을 표현할 때는 앞에 0을 접두사로 사용한다.)
(8) 16진수: %x 또는 %X를 사용한다.(값을 표현할 때는 앞에 0x를 접두사로 사용한다.)

▶ 열거형(enum): 관련된 이름에 대해 일련의 정수 값을 제공한다.(ex. enum 상수명 {값1,값2 ...};)

▶ 공용체(union): 같은 메모리 공간을 공유하지만 서로 다른 유형의 변수들을 표현할 수 있는 복합 데이터 타입이다.

▶ 배열(array)

: 배열은 자료형, 변수 이름, 배열의 크기, 변수의 값을 정해줌으로서 사용 가능하며, 배열의 인덱스는 0으로부터 시작한다.

```
int array[4] = {1,2,3,4}
array[0] = 1, array[1] = 2, array[2] = 3, array[3] = 4
                          < 1차원 배열의 사용 예시 >
int arr[2][3] = {{1,2,3},{4,5,6}}
arr[0][0] = 1, arr[0][1] = 2, arr[0][2] = 3
arr[1][0] = 4, arr[1][1] = 5, arr[1][2] = 6
                          < 2차원 배열의 사용 예시 >
```

▶ C 프로그래밍의 연산자 : C 언어는 여러 종류의 연산자를 지원한다.

(1) 산술 연산자(Arithmetic Operators) : + (덧셈), - (뺄셈), * (곱셈), / (나눗셈), % (나머지) 등이 있다.

(2) 비교 연산자(Comparison Operators) : == (같음), != (같지 않음), > (보다 큼), < (보다 작음), >= (크거나 같음), <= (작거나 같음) 등이 있다.

(3) 논리 연산자(Logical Operators) : &&(AND 연산자, 모든 조건이 참일 때 참 반환 ), ||(OR 연산자, 하나 이상 조건이 참일 때 참 반환 ), !(NOT 연산자, 조건이 거짓일 때 참 반환 )가 있다.

(4) 대입 연산자(Assignment Operators) : = , += , -= , *= , /= , %= 등이 있으며 주어진 값을 변수에 할당하거나 수정한다.

(5) 증가/감소 연산자(Increment/Decrement Operators) : ++ , -- 가 있으며 변수의 값을 1씩 증가 또는 감소시킨다.

- a++와 ++a의 차이점: 두 연산자 모두 a의 값을 1만큼 증가시켜준다. 그러나 두 연산자 사이에서는 중요한 차이점이 있다. a++는 후위 증가 연산자라고 부르며, 이 연산자는 먼저 a의 현재 값을 반환한 후에 a의 값을 1 증가시킨다. 반대로 ++a는 전위 증가 연산자라고 부르며, 이 연산자는 먼저 a의 값을 1 증가시킨 후에 증가된 값을 반환한다.

```
#include <stdio.h>
int main() {
    int a = 5;
    int b = a++; // a의 현재 값을 b에 할당하고, 그 후에 a의 값을 1 증가시킵니다.
    printf("a: %d, b: %d", a, b); // 출력: a: 6, b: 5
    a = 5;
    int c = ++a; // a의 값을 1 증가시키고, 그 후에 a의 값을 c에 할당합니다.
    printf("a: %d, c: %d", a, c); // 출력: a: 6, c: 6
    return 0;
}
```

< 전위 연산자와 후위 연산자의 차이 예시 >

(6) 비트 연산자(Bitwise Operators) : &(AND), |(OR), ^(XOR), ~(NOT), <<(left shift), >>(right shift) 등이 있다.

**2) 제어흐름**

▶ C 언어에서 제어 흐름은 프로그램의 실행 순서를 결정한다. C에서는 조건문, 반복문, 선택문 등을 통해 제어 흐름을 조작할 수 있다.

(1) 조건문: if, if-else, if-else if-else 등이 있다. 조건문은 주어진 조건이 참인지 거짓인지에 따라 프로그램의 흐름을 제어한다.

(2) 반복문: for, while, do-while 등이 있다. 반복문은 일정한 조건이 만족하는 동안 코드 블록을 반복하여 실행한다.

(3) 선택문: switch-case가 있다. 선택문은 변수의 값에 따라 프로그램의 흐름을 다양하게 제어할 수 있다.

(4) 점프문: break, continue, return, goto 등이 있다. 점프문은 프로그램의 흐름을 즉시 변경한다. break와 continue는 반복문 내에서 사용되며, 반복의 흐름을 제어한다. return은 함수의 실행을 종료하고, 호출자에게 제어를 반환한다. goto는 프로그램 내의 특정 위치로 제어를 이동시킨다.

**3) 함수와 프로그램 구조**

▶ 함수(Function)

: C 프로그래밍에서 함수는 특정 작업을 수행하는 코드의 블록이다. 함수를 사용하면 코드를 재사용하고 모듈화 할 수 있다, 즉, 프로그램을 더욱 이해하기 쉽고 관리하기 쉬운 부분으로 나눌 수 있다.

▶ 프로그램 구조

: C 프로그램은 기본적으로 하나 이상의 함수로 구성되며, 그 중 하나는 반드시 main이라는 이름의 특별한 함수여야 한다. C 프로그램 실행 시 가장 먼저 호출되는 것이 바로 이 main()함수이며, 여기서부터 모든 실행이 시작된다.

**4) 포인터와 배열**

▶ 포인터(pointer)

: 포인터란 메모리 공간의 주소이다. 따라서 포인터 상수는 메모리 주소 값이다. 포인터 변수는 주소 값(포인터)를 저장할 수 있는 변수로 다른 메모리 공간을 가리킨다.

- 포인터 변수의 선언

```
int a = 10;
int *p = &a;
```


그림 3.3 포인터 변수

< 포인터 변수의 선언 >

: 다음과 같이 선언하면 p는 정수형 변수를 가리키는 포인터 변수로 선언되고 변수 a의 주소 값으로 초기화된다. &는 번지 연산자로 변수의 주소를 나타낸다.

```
#include <stdio.h>
int main(){
   int a = 10;
   int *p = &a;
   printf("%lu : %d\n", &a, a);
   printf("%lu : %d\n", p, *p);
   *p = *p + 10; // a = a+10과 동일한 효과
   printf("%lu : %d\n", &a, a);
   printf("%lu : %d\n", p, *p);
   a = 30; //*p = 30과 동일
   printf("%lu : %d\n", &a, a);
   printf("%lu : %d\n", p, *p);
}
```

실행 결과

```
16054516 : 10
16054516 : 10
16054516 : 20
16054516 : 20
16054516 : 30
16054516 : 30
```

: *p는 포인터 p가 가리키는 곳을 따라가 데이터가 저장된 공간을 참조하는 것으로 이를 주소참조(dereferencing)라고 한다.

- 포인터 변수도 일종의 변수이므로 함수의 매개변수로도 사용될 수 있다.

```
void swap(int*px, int*py){
   int t;
   t = *px;
   *px = *py;
   *py = t;
}
```

```
int a = 10, b = 20;
swap(&a, &b);
printf("%d %d", a,b);
```

: 두 변수의 값을 상호 교환하는 swap함수는 포인터 변수 px와 py를 매개변수로 선언하였다. 이 함수를 호출할 때는 포인터(주소)를 인수로 전달해야 한다. 예를 들

어 위와 같이 변수 a와 b의 주소(포인터)를 인수로 전달하면 이 함수는 포인터 매개변수 px와 py를 통해 전달받은 포인터를 주소참조하여 해당 변수 a,b의 값을 상호교환 한다. 함수 호출 후에 변수 a,b의 값을 출력하여 이를 확인할 수 있다.

- 배열과 포인터

: 배열 이름은 실제로는 배열의 시작주소를 나타내는 상수 즉 포인터 상수이다. 따라서 포인터 변수에 배열 이름을 대입할 수 있다. 이 경우에 포인터 변수는 배열의 시작을 가리키게 된다.

```
#include <stdio.h>
int main(){
   int *p, a[4] = {10, 20, 30, 40};
   printf("%d %d %d %d\n", a[0], a[1], a[2], a[3]);
   p = a;
   printf("%d %d %d %d %d\n", *p, *(p+1), p[1], *(p+2), p[2]);
}
```

실행결과

```
10 20 30 40
10 20 20 30 30
```

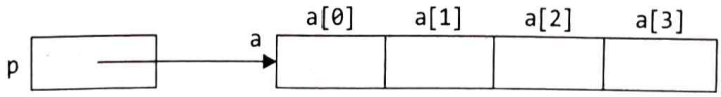



그림 3.4 배열과 포인터

< 배열과 포인터 예시 >

: 위와 같은 프로그램에서 포인터 변수 p에 배열 이름 a를 대입하면 포인터 변수 p는 위의 그림과 같이 배열 a의 시작부분을 가리킨다. 따라서 *p는 배열 a의 첫 번째 원소를 나타낸다. p+1은 p가 가리키는 배열 a에서 다음 원소의 주소를 나타낸다는 점을 유의해야 한다. 따라서 *(p+1)은 p가 가리키는 배열 a에서 다음 원소를 접근하게 된다. p[1]는 *(p+1)에 대한 일종의 단축 표현으로 같은 원소를 접근한다.

- 문자열과 포인터

: 문자열은 문자형(char) 배열(array)이라고 할 수 있다. 포인터 변수가 배열을 가리킬 수 있으므로 char 포인터 변수를 사용하여 문자열을 가리킬 수 있다.

```
char *p = “Hello”;
char m[] = “world”;
```

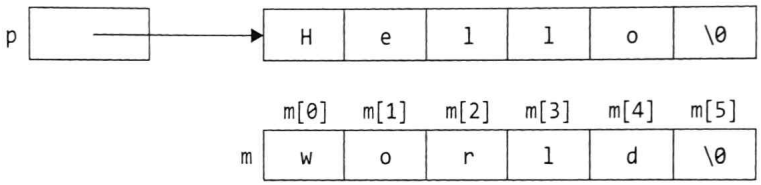



그림 3.5 포인터와 문자 배열

< 문자열과 포인터 예시 >

: 이 프로그램에서는 포인터 변수 p는 “Hello” 문자열을 가리키고, m은 문자 배열로 “world”로 초기화 되도록 선언하였다.

```
#include <stdio.h>
int main(){
   char *p = “Hello”;
   char m[] = “world”;
   printf(“%s %s\n”, p, m);
   p = m;
   printf(“%s\n”, p);
   while(*p)
      putchar(*p++);
}
```

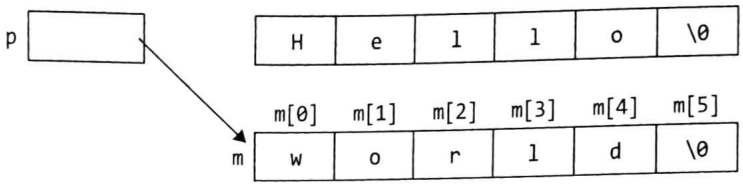



그림 3.6 p = m 수행 후 문자 배열에 대한 포인터

< 문자열과 포인터 예시 >

- 여러 개의 문자열 저장

```
char colors[3][10] = {“red”, “blue”, “white”};
char *ptr[3] = {“red”, “blue”, “white”};
```

: 여러 개의 문자열을 포인터 변수를 이용하여 저장하려면 위와 같이 선언해야 한다.

- 포인터의 포인터

```
char **p = ptr;
```

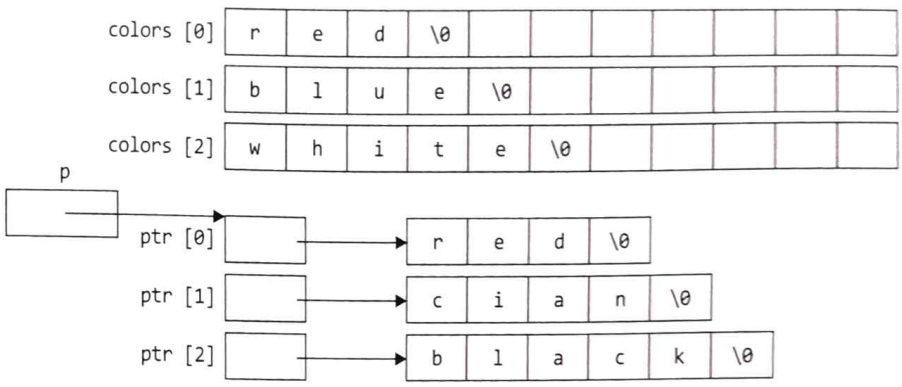



그림 3.7 이차원 배열과 포인터 배열

< 이중 포인터 >

: 포인터의 대상이 char 포인터인 포인터 변수를 사용할 수 있다. char 포인터의 포인터 변수 p는 포인터 배열 ptr의 시작 주소를 대입하여 포인터 배열 ptr을 가리킬 수 있다.

```
#include <stdio.h>

int main() {
    int num = 10;
    int *p = &num; // 포인터 p에 num의 주소를 저장합니다.
    int **pp = &p; // 이중 포인터 pp에 p의 주소를 저장합니다.
    printf("num's value: %d", num);
    printf("p's value: %p, points to: %d", p, *p);
    printf("pp's value: %p, points to: %p, points to: %d", pp, *pp,
**pp);
    return 0;
}
```

실행결과

num's value: 10

p's value: 주소값, points to: 10

pp's value: 주소값, points to: 주소값, points to: 10

: 위의 프로그램에서 p는 num의 주소를, pp는 p의 주소를 저장하고 있다.

▶ 배열(array)

: 배열은 자료형, 변수 이름, 배열의 크기, 변수의 값을 정해줌으로서 사용 가능하며, 배열의 인덱스는 0으로부터 시작한다.

```
int array[4] = {1,2,3,4}
array[0] = 1, array[1] = 2, array[2] = 3, array[3] = 4
                          < 1차원 배열의 사용 예시 >
int arr[2][3] = {{1,2,3},{4,5,6}}
arr[0][0] = 1, arr[0][1] = 2, arr[0][2] = 3
arr[1][0] = 4, arr[1][1] = 5, arr[1][2] = 6
                          < 2차원 배열의 사용 예시 >
```

**5) 구조**

▶ 구조체(struct): 서로 다른 타입의 변수들을 하나로 그룹화할 수 있는 복합 데이터 타입이다.

```
struct 구조체명 {
   자료형1 멤버변수1;
   자료형2 멤버변수2;
   ...
} [변수명 [= {초기값, ...}]];
                          < 새로운 구조체 선언 >
```

```
#include <stdio.h>

// 구조체 정의
struct Student {
    char name[50];
    int age;
    float grade;
};

int main() {
    // 구조체 변수 선언 및 초기화
    struct Student s1 = {"Kim", 20, 4.3};

    // 구조체 멤버 접근
    printf("Name: %s
", s1.name);
    printf("Age: %d
", s1.age);
    printf("Grade: %.2f
", s1.grade);

    // 구조체 멤버 수정
    s1.age = 21;
    printf("Age after modification: %d
", s1.age);

    return 0;
}
```

< 구조체 사용 예시 >

**6) 입력과 출력**

▶ C언어의 기본 입출력 함수

(1) printf(): 이 함수는 형식화된 문자열을 표준 출력에 출력한다. 다양한 형식 지정자를 사용하여 문자열, 정수, 실수 등을 원하는 형식으로 출력할 수 있다.

(2) scanf(): 이 함수는 형식화된 입력을 표준 입력으로부터 읽어 들인다. 다양한 형식 지정자를 사용하여 문자열, 정수, 실수 등을 원하는 형식으로 입력 받을 수 있다.
(3) fopen(): 이 함수는 파일을 열어 파일 포인터를 반환한다. 파일 모드에 따라 파일을 읽기, 쓰기, 추가 등의 용도로 열 수 있다.
(4) fclose(): 이 함수는 fopen() 함수에 의해 열린 파일을 닫는다.
(5) fprintf()와 fscanf(): 이들 함수는 각각 파일에 형식화된 출력을 하거나 파일로부터 형식화된 입력을 읽어 들인다.
(6) fgetc()와 fputc(): 이들 함수는 각각 파일로부터 한 글자를 읽어 들이거나 파일에 한 글자를 출력한다.
(7) gets(): 이 함수는 표준 입력으로부터 문자열을 읽어 들인다. 그러나 이 함수는 버퍼 오버플로우 문제를 일으킬 수 있으므로, 보안상의 이유로 사용을 권장하지 않는다. 대신 fgets() 함수를 사용하는 것이 좋다.
(8) puts(): 이 함수는 문자열을 표준 출력에 출력한다. 문자열의 끝에는 자동으로 개행 문자('\')가 추가된다.
(9) getch()와 getche(): 이들 함수는 사용자로부터 한 문자를 입력 받는다. getch()는 입력을 받을 때 화면에 표시하지 않고, getche()는 입력을 받을 때 화면에 표시한다. 이들 함수는 표준 C 라이브러리에는 없으며, 주로 Windows 환경에서 사용되는 함수들인다.
(10) putch(): 이 함수는 한 문자를 표준 출력에 출력한다. 이 함수 역시 표준 C 라이브러리에는 없으며, 주로 Windows 환경에서 사용되는 함수이다.
(11) getchar()와 putchar(): 이들 함수는 표준 C 라이브러리에서 제공하는 함수로, getchar()는 한 문자를 입력 받고, putchar()는 한 문자를 출력한다. 이들 함수는 getch(), getche(), putch()와 비슷한 역할을 하지만, 표준 C 라이브러리에서 제공되므로 모든 환경에서 사용할 수 있다.

**7) 컴파일러**

▶ 컴파일(Compile): 고급 언어로 작성된 프로그램 소스 코드를 컴퓨터가 이해할 수 있는 저급 언어, 즉 기계어로 변환하는 과정을 말한다. 이 작업은 컴파일러(Compiler)라는 특별한 소프트웨어에 의해 수행된다.

▶ 컴파일 과정

(1) 전처리(Preprocessing): 이 단계에서는 #include, #define 등의 전처리 지시자를 처리한다. 파일을 포함시키거나 매크로를 확장하는 등의 작업을 수행한다.
(2) 컴파일(Compilation): 이 단계에서는 소스 코드가 어셈블리어로 변환된다. 컴

파일러는 소스 코드의 문법을 검사하고, 문제가 없다면 어셈블리어로 변환한다.
(3) 어셈블(Assembly): 어셈블리 코드를 기계어 코드로 변환하는 단계이다. 결과물은 오브젝트 파일(Object file)이라고 부르며, 이 파일은 기계어 코드를 담고 있다.
(4) 링크(Linking): 여러 개의 오브젝트 파일과 필요한 라이브러리를 하나로 묶어 최종 실행 파일을 생성하는 과정이다. 이 단계에서는 외부 라이브러리의 함수를 참조하거나, 여러 소스 파일에서 정의된 함수를 서로 연결하는 등의 작업을 수행한다.
▶ 리눅스에서는 C 컴파일러로 gcc(GNU cc) 컴파일러가 널리 사용되고 있다. gcc 컴파일러는 대부분의 리눅스 배포판에 포함되어 있다.
▶ GCC(GNU Compiler Collection): GNU 프로젝트의 일부로, 여러가지 프로그래밍 언어를 지원하는 컴파일러 모음이다. C, C++, Objective-C, Fortran, Ada, Go 등의 언어를 지원하며, 그 외에도 다양한 하드웨어 아키텍처를 지원한다.
▶ GCC의 주요 구성 요소는 다음과 같다.
(1) 전처리기(Preprocessor): 전처리기는 소스 코드가 컴파일러에게 전달되기 전에 처리되는 단계이다. #include, #define 등의 전처리 지시자를 처리한다.
(2) 컴파일러(Compiler): 컴파일러는 전처리기를 통과한 소스 코드를 읽고, 이를 어셈블리 언어로 변환한다.
(3) 어셈블러(Assembler): 어셈블러는 컴파일러가 생성한 어셈블리 코드를 기계어로 변환한다.
(4) 링커(Linker): 링커는 여러 개의 오브젝트 파일을 하나의 실행 파일로 연결한다. 라이브러리 함수와 프로그램 코드를 결합하는 작업도 수행한다.
▶ 단일 모듈 프로그램 컴파일
- 단일 모듈 프로그램: 단일 모듈 프로그램은 하나의 파일로 이루어진 C 프로그램으로 간단한 프로그램인 경우에 하나의 모듈(파일)로 작성할 수 있다.
- gcc를 이용한 단일 모듈 프로그램 컴파일

```
$ gcc [-옵션] 파일
C 프로그램을 컴파일한다. 옵션을 사용하지 않으면 실행 파일 a.out를 생성한다.
```

· -c 옵션: 이 옵션을 사용하여 목적 파일을 만들 수 있다. 목적 파일(object file)은 컴파일은 되었지만 링크되지 않아 아직 실행될 수 없는 파일이다.
· -o 옵션: 이 옵션을 사용하여 실행 파일(executable file) 이름을 지정하여 만들 수 있다. 예를 들어 longest.c라는 파일을 컴파일한다고 해보자.

```
$ gcc –o longest longest.o
$ longest
```
```
$ gcc –o longest longest.c
```

: 위의 첫 번째 예시와 같이 목적 파일로부터 지정한 이름의 실행 파일을 만들 수 있고, 두 번째 예시와 같이 원시 파일로부터 파로 실행 파일을 만들 수도 있다.

· -S 옵션: 이 옵션을 사용하여 어셈블리어 프로그램 longest.s 파일을 생성할 수 있다.

· -l 옵션: 이 옵션을 사용하여 특정 라이브러리를 링크할 수 있다. -lxxx는(보통 /usr/lib 디렉토리에 있는) 라이브러리 libxxx.a를 링크하라는 의미이다. 예를 들어 test.c 프로그램이 수학 라이브러리(libm.a)를 사용한다면 –lm옵션을 이용하여 libn.a를 링크할 수 있다.

```
$ gcc –o test –lm test.c
```

▶ 다중 모듈 프로그램 컴파일

- 다중 모듈 프로그램: 프로그램의 코드를 여러 개의 파일 또는 모듈로 분리하여 작성하는 방식을 말한다. 각 모듈은 프로그램의 특정 기능을 담당하며, 모듈 간에는 정의된 인터페이스를 통해 데이터를 주고받는다. 다중 모듈 프로그램은 .h확장자를 가지는 헤더 파일과 프로그램 파일 등의 여러 파일을 포함한다.

- gcc를 이용한 다중 모듈 프로그램 컴파일

```
$ gcc –c main.c
$ gcc –c copy.c
$ gcc –o main main.o copy.o
```
```
$ gcc –o main main.c copy.c
$ main
```

: 각 소스 파일을 각각 컴파일하고 그들을 링크하여 실행 파일을 만드는 것이 기본적인 방법이다. 각 소스 코드 파일을 개별적으로 컴파일하기 위해서는 첫 번째 예시와 같이 gcc의 –c 옵션을 사용하면 되는데 각 소스 코드 파일에 대해 .o 확장자를 갖는 목적 파일이 생성된다. 생성된 목적 파일들을 gcc의 –o 옵션을 이용하여 링크하여 실행 파일을 생성하면 된다. 혹은 두 번째 예시와 같이 단번에 하고 실행할 수도 있다.

▶ 네이티브 컴파일러(Native Compiler)와 크로스 컴파일러(Cross Compiler)

(1) 네이티브 컴파일러(Native Compiler): 네이티브 컴파일러는 소스 코드를 같은 시스템에서 실행될 기계어로 컴파일한다. 즉, 컴파일러가 실행되는 시스템과 목표 시스템이 동일하다. 예를 들어, Windows 시스템에서 Windows 애플리케이션을

컴파일하려면 Windows용 네이티브 컴파일러를 사용한다.

(2) 크로스 컴파일러(Cross Compiler): 크로스 컴파일러는 소스 코드를 다른 시스템에서 실행될 기계어로 컴파일한다. 즉, 컴파일러가 실행되는 시스템과 목표 시스템이 다르다. 예를 들어, Windows 시스템에서 Linux 애플리케이션을 컴파일하려면 Linux용 크로스 컴파일러를 사용한다.

### 8) 자동 빌드 도구(Make 시스템)

▶ 빌드(Build): 소프트웨어 개발 과정에서 소스 코드를 실행 가능한 소프트웨어 프로그램으로 변환하는 과정을 의미한다. 이 과정은 일반적으로 컴파일, 링크 등의 단계를 포함하며, 이를 통해 소스 코드가 최종적인 실행 파일, 라이브러리, 패키지 등의 형태로 생성된다.

▶ make 시스템의 필요성

: 다중 모듈 프로그램을 컴파일하여 실행 파일을 만들기 위해 빌드하는 과정은 번거로울 수 있다. 예를 들어, 여러 파일 중 copy.c라는 소스 코드만 수정하고자 한다면, 이 파일을 컴파일하여 목적 파일 copy.o를 생성하고 이 목적 파일을 사용한 다른 목적 파일과 다시 링크하여 실행파일을 생성해야 한다. 간단한 프로그램에서는 쉽지만, 대규모 프로그램에서는 여러 파일 간의 관계를 기억하고 있기가 쉽지가 않다. make 시스템을 이용하면 이 과정을 효과적으로 할 수 있다.

▶ make 시스템

: make 시스템을 사용하여 실행 파일을 관리하기 위해서는 먼저 메이크파일을 만들어야 하는데 이 파일에 실행 파일을 만들기 위해 필요한 파일들과 그들 사이의 의존 관계, 만드는 방법 등을 기술한다. make 시스템은 메이크파일을 이용하여 파일의 상호 의존 관계를 파악하여 실행 파일을 쉽게 다시 만들 수 있다.

```
$ make [-f 메이크파일]
```
make 시스템은 메이크파일(makefile 혹은 Makefile)을 이용하여 보통 실행 파일을 빌드한다. 옵션을 사용하여 별도의 메이크파일을 지정할 수 있다.

: 다음과 같이 make 시스템을 실행하면 메이크파일 내의 일련의 의존 규칙들에 근거하여 실행 파일을 최신 버전으로 개정한다. -f 옵션을 이용하여 make 시스템의 대상(입력)이 되는 메이크파일 이름을 명시할 수 있으며 별도로 명시하지 않으면 Makefile 혹은 makefile을 사용한다.

▶ Makefile(makefile)

: 메이크파일은 실행 파일을 만들기 위해서 사용되는 파일들 사이에 상호 의존 관계 및 실행 파일을 만드는 방법 등을 기술한다. 이 파일은 어떠한 이름이라도 가질 수 있지만 보통 Makefile 혹은 makefile이라는 이름으로 만든다. 메이크파일의 일

반적인 구성 형식은 다음과 같다.

```
목표(target): 의존리스트(dependencies)
              명령리스트(commands)
```

- 목표(target): 생성하고자 하는 목표 파일(일반적으로 목적 파일 혹은 실행 파일) 작업의 이름이다.
- 의존리스트(dependencies): target을 빌드하기 위해 필요한 파일들을 지정하고 이 파일들 중 하나라도 변경되면, target이 다시 빌드된다.
- 명령리스트(commands): target을 빌드하기 위해 실행해야 하는 명령들을 지정한다. 각 명령은 새로운 줄에 적고, 탭 문자로 시작해야 한다.

Makefile

```
main: main.o copy.o
        gcc –o main main.o copy.o
main.o: main.c copy.h
          gcc –c main.c
copy.o: copy.c
          gcc –c copy.c
```

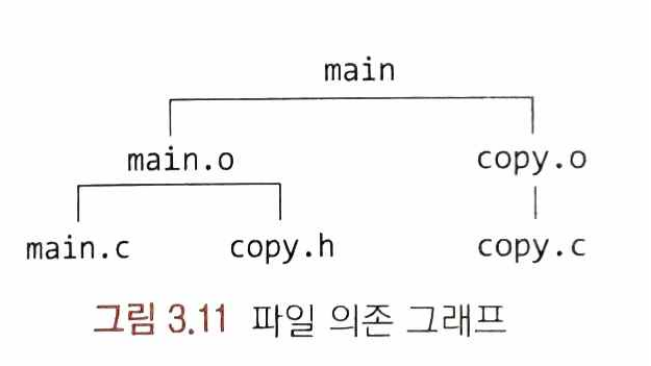



그림 3.11 파일 의존 그래프

< Makefile 파일 의존 그래프 >

: 메이크파일은 위와 같은 예시와 같이 구성할 수 있다.

▶ 매크로(Macro)

: 프로그래밍에서 반복되는 코드나 복잡한 코드를 단순화하고 재사용하기 위한 방법이다. 매크로는 일반적으로 특정 코드 패턴을 대체하는 단순한 문자열 치환을 수행하지만, 일부 고급 매크로 시스템에서는 복잡한 코드 생성 및 변환을 수행할 수 있다.

▶ 매크로 정의

: Makefile에서 매크로는 변수와 유사한 개념으로 사용된다. 즉, 매크로는 특정 값을 저장하고, 이 값을 나중에 재사용할 수 있게 해주는 역할을 한다.

| MACRO = value |
| --- |
| $(MACRO) |

: 이와 같이 메이크파일에서 매크로를 정의하고 정의된 매크로를 사용할 수 있다.

▶ 매크로 작성 기본 규칙

- 매크로의 정의는 '='를 포함하는 하나의 문장이어야 한다.
- '#'은 주석문의 시작이다.
- 여러 행을 사용할 때는 '\'를 사용한다.
- 매크로를 참조할 때는 괄호나 중괄호를 둘러싸고 앞에 '$'를 붙인다.
- 정의되지 않은 매크로를 참조할 때는 NULL 문자열로 치환된다.

### 9) 링커(Linker)와 로더(Loader)

▶ 링커(Linker): 컴파일러가 생성한 여러 개의 오브젝트 파일을 결합하여 하나의 실행 가능한 프로그램을 만드는 역할을 한다. 링커는 라이브러리 함수의 참조를 해결하고, 메모리에서 프로그램의 각 부분이 위치할 주소를 결정하는 등의 작업을 수행한다. 링커는 정적 링킹과 동적 링킹을 수행할 수 있다.

▶ 로더(Loader): 로더는 링커가 생성한 실행 가능한 프로그램을 메모리에 로드하여 실행할 수 있게 하는 역할을 한다. 로더는 프로그램의 코드와 데이터를 메모리의 적절한 위치에 배치하고, 프로그램의 시작 지점으로 제어를 전달함으로써 프로그램을 실행한다. 또한, 프로그램 실행 중에 필요한 동적 라이브러리를 로드하는 역할도 한다.

### 10) 정적 링킹(Static Linking)과 동적 링킹(Dynamic Linking)

▶ 정적 링킹(Static Linking): 실행 가능한 목적 파일을 만들 때 프로그램에서 사용하는 모든 라이브러리 모듈을 복사하는 방식을 말하며 링커에 의해 이루어진다. 즉, 자신이 작성한 프로그램에서 A라는 외부 함수를 사용했다면, A라는 외부 함수에 대한 정보를 자신이 작성한 프로그램의 실행파일을 만들 때 복사해온다.

- 장점: 정적 링킹 라이브러리를 사용하는 프로그램은 동적 링킹 라이브러리를 사용하는 프로그램보다 빠르다. 또한 정적 링킹 프로그램에서 모든 코드는 하나의 실행 모듈에 담기기 때문에 compatibility issues 즉, 불일치에 대한 걱정을 하지 않아도 된다.
- 단점: 5개의 프로그램에서 A라는 외부 함수를 이용하는데 이때 정적 링킹 방식을 사용하면 5개의 프로그램의 실행 가능한 목적파일 각각에 A의 정보가 담긴다. 즉, 중복이 발생한다. 따라서 정적 링킹으로 만들어진 프로그램은 크기가 크고 메모리 효율이 좋지 않다. 또한 정적 링킹을 이용하면, A라는 함수에 변화가 생길 경우

그 변화를 적용하기 위해서 다시 컴파일하여 다시 링킹을 해야만 한다. 실행 가능한 목적파일을 만들 때 A에 관한 정보를 그냥 복사해왔기 때문에 A에 변화가 생겨도 다시 컴파일하지 않는 이상 변화가 적용되지 않는다.

▶ 동적 링킹(Dynamic Linking): 동적 링킹이란 실행 가능한 목적 파일을 만들 때 프로그램에서 사용하는 모든 라이브러리 모듈을 복사하지 않고 해당 모듈의 주소만을 가지고 있다가, 런타임에 실행 파일과 라이브러리가 메모리에 위치될 때 해당 모듈의 주소로 가서 필요한 것을 들고 오는 방식이다. 런타임에 운영체제에 의하여 이루어진다.

- 장점: 동적 링킹 방식을 이용하면 5개의 프로그램에서 A라는 외부 함수를 이용한다고 해도 A라는 함수의 정보는 하나만 있으면 된다. 각각의 실행 가능한 목적파일에서는 A를 복사해서 A 그 자체를 가지고 있는 것이 아니라 A가 있는 곳의 주소만 가리키고 있기 때문이다. 따라서 정적 링킹 방식에 비해 실행 가능한 목적파일의 크기가 작다. 즉, 메모리와 디스크 공간을 더 아낄 수 있다.

동적 링킹을 이용하면, A라는 함수에 변화가 생겨도 그 변화를 적용하기 위해 다시 컴파일하여 다시 링킹할 필요가 없다. 실행 가능한 목적파일을 만들 때 A에 관한 정보를 그냥 복사해온 것이 아니라 A가 있는 곳의 주소를 담았기 때문이다. 그저 가리키는 곳을 따라가면 변화된 A가 있다.

- 단점: 동적 링킹 방식은 정적 링킹 방식보다 느리다. 매번 주소를 따라가야하는 오버헤드가 존재하기 때문이다. 또한 동적 링킹 방식은 compatibility issues 즉, 불일치에 대한 문제를 고려해야 한다. 왜냐하면 동적 링킹 방식은 불일치가 존재할 수 있기 때문이다.

**11) 개발 환경**

▶ 로컬 개발 환경

: 로컬 개발 환경은 개발자의 개인 컴퓨터나 작업 스테이션 위에서 구축되는 환경을 말한다. 이 환경에서 개발자는 코드를 작성하고, 컴파일하고, 실행하며, 필요한 경우 디버깅을 수행한다. 로컬 개발 환경은 개발자가 자유롭게 실험하고 변경을 시도할 수 있는 공간을 제공한다.

▶ 통합 개발 환경(IDE)

: 통합 개발 환경은 코드 편집, 컴파일, 디버깅, 버전 관리 등의 기능을 한 곳에서 제공하는 소프트웨어이다. Visual Studio, Eclipse, IntelliJ IDEA 등이 이에 해당한다. IDE를 사용하면 개발 과정이 단순화되고 효율성이 향상될 수 있다.

▶ 교차 개발 환경(Cross Development Environment)

: 한 종류의 시스템에서 다른 종류의 시스템을 위한 소프트웨어를 개발할 수 있도

록 하는 개발 환경을 말한다. 이를 위해 교차 컴파일러(Cross Compiler)와 같은 도구들이 사용된다.

▶ 클라우드 기반 개발 환경

: 클라우드 기반 개발 환경은 개발 도구와 리소스를 인터넷을 통해 제공하는 환경이다. 이 환경을 사용하면 개발자는 어디서나 작업을 수행할 수 있으며, 하드웨어 리소스에 대한 걱정 없이 개발을 진행할 수 있다. AWS Cloud9, Google Cloud Shell 등이 이에 해당한다.

## 2. 객체지향 프로그래밍

### 1) 객체지향원리

▶ 객체지향 프로그래밍(Object-Oriented Programming, OOP)은 컴퓨터 프로그래밍의 패러다임 중 하나로, 소프트웨어를 객체들의 모임으로 간주하고 이들 간의 상호작용으로 프로그램을 구현하는 방식이다.

▶ 클래스: 어떤 문제를 해결하기 위한 데이터를 만들기 위해 추상화를 거쳐 집단에 속하는 속성과 행위를 변수와 메서드로 정의한 것으로 객체를 만들기 위한 메타 정보라고 볼 수 있다.

▶ 인스턴스(객체): 클래스에서 정의한 것을 토대로 실제 메모리에 할당된 것으로 실제 프로그램에서 사용되는 데이터이다.

▶ 객체: 객체지향 프로그래밍에서 객체는 데이터와 그 데이터를 처리하는 함수(메서드)를 하나로 묶은 것을 의미한다. 객체는 클래스라는 틀에서 생성되며, 이 클래스는 객체의 특성(속성)과 행동(메서드)을 정의한다. 객체는 동일한 클래스를 기반으로 여러 개 생성될 수 있으며, 각각의 객체는 서로 다른 상태를 가질 수 있다.

예를 들어, "자동차"라는 클래스를 생각해보면, 이 클래스는 속성으로 "색상", "브랜드", "모델" 등을 가질 수 있고, 행동으로는 "운전하기", "정차하기" 등의 메서드를 가질 수 있다.

이 "자동차" 클래스를 기반으로 실제 "자동차" 객체를 생성하면, 각 객체는 자신만의 속성 값을 가질 수 있다. 예를 들어, 한 객체는 "빨간색", "테슬라", "모델3"의 속성을 가질 수 있고, 다른 객체는 "흰색", "현대", "소나타"의 속성을 가질 수 있다.

▶ 객체지향 프로그래밍의 주요 원칙

(1) 캡슐화(Encapsulation): 캡슐화는 데이터와 그 데이터를 조작하는 함수를 하나로 묶는 과정을 의미한다. 이렇게 만들어진 '객체'는 자신의 내부 데이터를 직접적

으로 접근할 수 없도록 숨기고(정보 은닉), 대신 함수를 통해서만 조작할 수 있도록 한다. 이런 방식은 코드의 안정성과 유지보수성을 향상시킨다.

(2) 상속(Inheritance): 상속은 한 클래스가 다른 클래스의 속성과 메소드를 물려받아 사용할 수 있게 하는 것이다. 이를 통해 코드 재사용이 가능하며 계층적인 클래스 구조를 만들 수 있다.

(3) 다형성(Polymorphism): 다형성은 같은 이름의 메소드가 서로 다른 기능을 수행하도록 하는 원칙이다. 예를 들어 부모 클래스에서 정의된 메소드가 자식 클래스에서 오버라이딩되어 다르게 동작할 수 있다. 또한 인터페이스나 추상 클래스 등을 활용하여 여러 타입에 대해 동일한 작업을 정의하는 것도 다형성에 포함된다.

(4) 추상화(Abstraction): 추상화는 복잡한 시스템을 단순한 인터페이스로 나타내는 것이다. 사용자가 필요로 하는 정보와 기능만 제공하며 내부 구현에 대해서는 숨긴다. 이것 역시 코드 관리와 안정성 향상에 도움이 된다.

▶ 객체지향 프로그래밍의 장점과 단점

(1) 장점

- 코드 재사용이 용이하다. 기존에 만들어져 있는 클래스를 가져와서 이용할 수 있고, 상속을 통해 확장해서 사용할 수 있다.
- 유지보수가 쉽다. 절차 지향 프로그래밍에서는 코드를 수정해야할 때 일일이 찾아 수정해야 하는 반면 객체 지향 프로그래밍에서는 수정해야 할 부분이 클래스 내부에 멤버 변수 혹은 메서드로 존재하기 때문에 해당 부분만 수정하면 된다.
- 대형 프로젝트에 적합하다. 클래스 단위로 모듈화시켜서 개발할 수 있으므로 대형 프로젝트처럼 여러 명, 여러 회사에서 프로젝트를 개발할 때 업무 분담하기 쉽다.

(2) 단점

- 처리 속도가 상대적으로 느리다.
- 객체가 많으면 용량이 커질 수 있다.
- 설계할 때 많은 시간과 노력이 요구된다.

### 2) C++개요

▶ C++는 절차 프로그래밍을 지원함과 동시에 객체지향 프로그래밍 언어로, 클래스와 객체를 통해 데이터와 함수를 하나의 단위로 묶는 캡슐화, 클래스 간의 관계를 정의하는 상속, 그리고 다형성 등을 지원한다.

(1) 클래스와 객체: C++에서 클래스는 데이터 멤버와 멤버 함수로 구성되며 이들을 하나의 단위로 묶는다. 이렇게 정의된 클래스를 바탕으로 실제 메모리에 할당된 인스턴스가 바로 객체이다. 예를 들어, 학생이라는 클래스가 있으면 이 학생 클래스에 이름, 나이 등의 속성(데이터 멤버)과 공부한다, 잔다 등의 동작(멤버 함수)을

정의할 수 있다.

(2) 캡슐화: 캡슐화는 관련된 데이터와 함수를 하나로 묶는 것이다. 즉, 데이터(멤버 변수)와 그 데이터를 조작하는 방법(멤버 함수)을 하나의 '클래스'라는 캡슐 안에 넣어 정보를 은닉하고 보호한다.

(3) 상속: 상속은 기존 클래스에서 속성과 메소드를 물려받아 새로운 클래스를 생성하는 것이다. 이 때 기존에 있는 클래스를 부모 혹은 슈퍼클래스라고 하고, 상속을 통해 생성된 새로운 클래스를 자식 혹은 서브클래스라고 한다.

(4) 다형성: 다형성은 한 가지 형태가 여러 가지 작업을 수행할 수 있도록 하는 원칙이다. 오버라이딩(메소드 재정의), 오버로딩(함수 중복정의) 및 가상 함수(virtual function) 등을 통해 구현된다.

(5) 추상화: 추상화란 복잡한 시스템에서 중요한 관점만을 모아 복잡도를 줄이는 프로그래밍 기법이다.

(6) 생성자(Constructor) 와 소멸자(Destructor): 생성자는 객체가 생성될 때 호출되어 초기 작업을 수행하며, 소멸자는 객체가 파괴될 때 호출되어 마무리 작업을 수행한다.

(7) 연산자 오버로딩: C++에서는 연산자를 재정의하여 사용자 정의 타입에서도 기본 타입처럼 동작하게 할 수 있다.

▶ 라이브러리 등 대부분의 자료 구조와 사용 방법은 C와 같다.

### 3) C++ 객체지향기능

▶ 위의 C++ 개요 내용과 같다.

### 4) Java 개요

▶ Java는 1995년에 Sun Microsystems(현재 Oracle)에서 개발된 객체지향 프로그래밍 언어이다. Java는 "한 번 작성하면 어디에서나 실행할 수 있다(Write Once, Run Anywhere)"는 원칙을 가지고 있으며, 이를 가능하게 하기 위해 자바 가상 머신(JVM)이라는 중간 단계를 거치도록 설계되었다.

▶ Java 주요 특징

(1) 객체 지향: Java는 순수한 객체 지향 언어로서 클래스와 객체, 상속, 다형성 등의 기본적인 객체 지향 원칙을 모두 지원한다.

(2) 플랫폼 독립성: Java 애플리케이션은 JVM 위에서 동작하기 때문에 운영체제나 하드웨어가 변경되더라도 동일한 코드를 재사용할 수 있다.

(3) 자동 메모리 관리(Garbage Collection): Java는 가비지 컬렉터를 통해 자동으로 메모리를 관리한다. 이로 인해 개발자는 메모리 할당 및 해제에 대해 신경 쓰

지 않아도 된다.
(4) 다중 스레드 지원: Java API에 내장된 스레드 기능을 사용하여 복잡한 멀티스레딩 프로그램을 쉽게 작성할 수 있다.
(5) 보안: JVM은 바이트코드 검증과 클래스 로딩 과정에서 보안 검사를 수행함으로써 시스템의 보안 유지에 도움을 준다.
(6) 네트워크 프로그래밍과 분산 컴퓨팅 지원: TCP/IP 네트워크 프로토콜 라이브러리와 HTTP, FTP 등의 인터넷 기반 서비스 처리 라이브러리가 포함되어 있다.
(7) 런타임 예외 처리: 컴파일 타임 에러뿐만 아니라 런타임 에러까지 체크하는 강력한 예외 처리 시스템을 제공한다.
(8) 다양한 API 제공: 데이터베이스 연결, 날짜/시간 처리, 파일 I/O 등 다양한 기능을 제공하는 풍부한 표준 API를 제공한다.
▶ Java 주요 개념
(1) 변수와 데이터 타입: Java에서도 변수를 선언할 때 해당 변수의 데이터 타입을 명시해야 한다. 기본 데이터 타입으로는 int(정수), double(실수), char(문자), boolean(불리언) 등이 있다.
(2) 연산자: Java는 다양한 연산자를 제공한다. 산술 연산자(+, -, *, /, %), 비교 연산자(==, !=, <, >, <=, >=), 논리 연산자(&&, ||, !), 대입 연산자(=), 증감 연산자(++, --) 등을 사용할 수 있다.
(3) 제어문: Java는 if, else, switch와 같은 조건문과 for, while, do-while과 같은 반복문을 제공한다. 이들을 이용해 프로그램의 흐름을 제어할 수 있다.
(4) 함수(메서드): Java에서 함수는 메서드라고 부른다. 메서드는 특정 작업을 수행하는 코드의 묶음으로, 클래스 내부에 선언된다.
(5) 배열과 문자열: 배열은 동일한 타입의 데이터를 연속적으로 저장하는 데이터 구조이다. 문자열은 Java에서 기본적으로 String 클래스를 사용해 표현한다.
(6) 클래스와 객체: Java는 순수한 객체 지향 프로그래밍 언어이다. 클래스는 데이터와 메서드를 하나로 묶는 설계도이며, 객체는 클래스의 인스턴스이다.
(7) 상속과 다형성: Java의 객체 지향 특징 중 하나로, 클래스 간의 관계를 정의하고 코드의 재사용성을 높인다. 인터페이스를 통한 다형성도 중요한 개념이다.
(8) 예외 처리: Java에서는 예외 처리를 통해 프로그램의 안정성을 높인다. try, catch, finally, throw, throws 등의 키워드를 이용해 예외를 처리한다.
(9) Java API: Java는 풍부한 API를 제공한다. 이를 이용해 파일 입출력, 멀티스레딩, 네트워킹, 데이터베이스 접근 등 다양한 작업을 수행할 수 있다.
▶ 오버라이딩(Overriding)과 오버로딩(Overloading)
(1) 오버라이딩(Overriding): 오버라이딩이란 상위 클래스 또는 인터페이스에서 이

미 제공하는 메서드를 하위 클래스에서 재정의하는 것을 말한다. 오버라이딩은 주로 상속 관계에서 사용되며, 하위 클래스가 상위 클래스의 메서드를 자신의 필요에 맞게 변경할 수 있게 해준다. 메서드를 오버라이딩하면, 그 메서드의 이름, 파라미터, 반환 타입은 동일하게 유지되지만, 그 메서드의 본체(동작 방식)는 하위 클래스에서 재정의된다.

예를 들어, '동물'이라는 클래스에 '소리내기'라는 메서드가 있을 때, '고양이'와 '강아지'라는 하위 클래스에서 이 메서드를 오버라이딩하여 각각 "야옹", "멍멍"이라는 소리를 출력하도록 변경할 수 있다.

(2) 오버로딩(Overloading): 오버로딩이란 같은 이름의 메서드나 함수를 여러 개 정의하되, 그들의 파라미터(인자)의 수나 타입을 다르게 하는 것을 말한다. 오버로딩은 같은 기능을 하는 메서드나 함수가 다양한 타입의 인자를 받을 수 있도록 해준다. 메서드를 오버로딩하면, 그 메서드의 이름은 동일하게 유지되지만, 파라미터의 수나 타입이 다르게 된다.

예를 들어, '곱하기'라는 함수를 오버로딩하여 두 개의 정수를 받는 함수와 두 개의 실수를 받는 함수를 만들 수 있다. 이렇게 하면 '곱하기'라는 이름으로 두 가지 타입의 인자를 모두 처리할 수 있게 된다.

**5) Java 객체지향기능**

(1) 클래스와 객체: Java에서 클래스는 데이터(필드)와 기능(메소드)을 하나로 묶은 설계도이다. 이 클래스를 바탕으로 실제 인스턴스를 생성하면 그것이 바로 객체이다.

(2) 캡슐화: 캡슐화는 데이터와 메소드를 하나의 클래스라는 단위로 묶고, 외부에서 직접 접근할 수 없도록 정보를 은닉하는 것이다. Java에서는 접근 제어자(private, protected, public 등)를 통해 캡슐화를 구현한다.

(3) 상속: 상속은 기존에 정의된 클래스의 필드와 메소드를 물려받아 새로운 클래스를 생성하는 것이다. 이렇게 상속을 통해 코드 재사용성이 향상되고 계층적인 구조가 형성된다.

(4) 다형성: 다형성은 같은 타입이거나 같은 계층구조에 있는 여러 객체가 같은 메시지에 대해 각각 다르게 반응하도록 하는 것이다. Java에서는 오버라이딩과 인터페이스 등을 활용하여 다형성을 구현할 수 있다.

(5) 추상화: 추상화는 복잡한 시스템을 간단하게 표현하는 방법으로, 중요한 부분만 강조하고 나머지 부분은 숨김으로써 복잡도를 줄인다. Java에서는 abstract 키워드 및 인터페이스 등을 사용하여 추상화를 구현할 수 있다.

(6) 인터페이스: 인터페이스란 메서드들의 집합체로서 해당 메서드들의 명세만 제공

하며 실제 구현 내용은 없다.
(7) 생성자(Constructor): 생성자란 객체가 생성될 때 자동으로 호출되어 초기 작업을 수행하는 메소드이다.
(8) 예외 처리(Exception Handling): 예외 처리는 프로그램 실행 중 발생할 수 있는 오류 상황을 미리 예측하고 적절히 대응하기 위한 기능이다.

## 3. 멀티미디어 프로그래밍

### 1) 멀티미디어 정보표현

▶ 멀티미디어 정보를 표현하는 방법은 주로 사용되는 미디어 유형인 텍스트, 이미지, 오디오, 비디오에 따라 다르게 적용된다. 각 미디어 유형을 디지털 환경에서 어떻게 표현하고 저장하는지에 대한 개념을 이해하는 것이 중요하다.
(1) 텍스트: 컴퓨터에서 텍스트는 ASCII나 Unicode와 같은 문자 인코딩 체계를 사용하여 숫자로 변환되어 저장된다. HTML과 같은 마크업 언어를 사용하면 텍스트의 구조와 스타일을 정의할 수 있다.
(2) 이미지: 이미지는 비트맵(bitmap)과 벡터(vector) 두 가지 주요 형식으로 나눌 수 있다. 비트맵 이미지는 각각의 픽셀이 고유한 색상 값을 가진 그리드로 구성되며, JPEG, GIF, PNG 등 다양한 파일 형식으로 저장된다. 반면 벡터 이미지는 수학적 함수와 기하학적 형태를 사용하여 이미지를 정의하므로 확대 및 축소 시에도 해상도가 유지된다.
(3) 오디오: 오디오 데이터는 일반적으로 아날로그 신호를 디지털 데이터로 변환하여 저장한다. 이 변환 과정에서 샘플링 속도(초당 샘플 수), 비트 깊이(샘플 당 비트), 채널 수(모노, 스테레오 등) 등이 중요한 역할을 한다.
(4) 비디오: 비디오는 연속된 이미지 프레임들의 시퀀스이다. 각 프레임은 개별적인 이미지와 같이 처리되며 프레임 속도(frame rate), 해상도(resolution), 압축 방식(compression method) 등이 중요하다.
(5) 애니메이션: 애니메이션은 일련의 2D 또는 3D 이미지가 시간에 따라 변화하는 것을 보여주기 위해 생성된다.
▶ 각각의 멀티미디어 요소들은 원시 데이터(raw data) 형태 그대로 많은 저장 공간을 차지할 수 있으므로, 적절한 압축 기법이 필요하다. 이를 위해 여러 가지 표준화된 압축 방식과 파일 형식이 사용된다.
멀티미디어 프로그래밍에서는 이러한 다양한 멀티미디어 요소들을 조작하고, 사용자 인터랙션에 반응하도록 하는 코드를 작성한다. 이는 게임 개발, 웹 개발, 모바일 앱

개발 등 다양한 분야에서 중요한 역할을 한다.

**2) 멀티미디어 압축**

▶ 멀티미디어 데이터는 텍스트, 이미지, 오디오, 비디오 등 다양한 형태의 데이터를 포함하며, 이들은 원시 형태로 저장되면 매우 큰 용량을 차지한다. 따라서 효율적인 저장 및 전송을 위해 압축이 필요하다.

▶ 멀티미디어 압축에는 주로 두 가지 방법인 '손실 압축'과 '비손실 압축'이 사용된다.

(1) 비손실 압축: 비손실 압축은 원본 데이터를 완전히 복구할 수 있는 방식의 압축인데, 이 방식은 데이터의 정확성이 중요한 경우에 사용된다. 예를 들어 텍스트 파일이나 중요한 문서 등에서 주로 사용된다. 하지만 멀티미디어 콘텐츠에서는 대부분 손실형인데 그 이유는 소량의 정보 손실이 있더라도 인간 감각에 크게 영향을 미치지 않으면서 압축이 가능하기 때문이다.

(2) 손실 압축: 손실압축에서 일부 정보가 제거되거나 변경되므로 원본과 완전히 동일하게 복구할 수 없다. 그러나 대신 더 높은 압축비율을 얻을 수 있다. JPEG(이미지), MP3(오디오), MPEG(비디오) 등 많은 멀티미디어 파일에서 사용된다.

▶ 각각의 멀티미디어 유형마다 적합한 다양한 표준화된 파일과 코덱(codec)들이 있다.

(1) 텍스트: 텍스트 파일 자체는 이미 많은 공간을 차지하지 않기 때문에 일반적으로 추가적인 컴패션 없이 ASCII나 Unicode와 같은 문자 인코딩 체계를 사용하여 저장된다.

(2) 그래프(비트맵 이미지): PNG는 비손실 압축비율을 제공하는 반면 JPEG는 고효율의 속실 압축비율을 제공한다.

(3) 오디오: 오디오 파일에서는 MP3, AAC 등이 널리 사용된다. 이들은 손실 압축비율을 사용하여 대체로 우수한 사운드 퀄리티를 유지하면서 파일 크기를 크게 줄인다.

(4) 비디오: 비디오에는 MPEG-4, H.264, H.265/HEVC 등이 있다. 이들은 공간적(프레임 내) 및 시간적(프레임 간) redundency를 제거하여 높은 압축비율을 달성한다.

**3) 영상 및 신호 처리**

▶ 멀티미디어 프로그래밍에서 영상 및 신호 처리는 매우 중요한 부분을 차지한다. 이는 디지털 이미지와 비디오, 오디오 등의 데이터를 처리하고 분석하는 데 사용되는 일련의 기술과 알고리즘을 포함한다.

(1) 영상 처리: 영상 처리는 디지털 이미지에 대한 다양한 연산을 수행하는 기술이다. 이에는 필터링, 색 조정, 명암 대비 조정, 잡음 제거, 에지 감지 등이 포함된다. 또한 더 복잡한 작업들도 가능하며, 예를 들어 객체 인식(object recognition), 얼굴 인식(face recognition), 모션 추적(motion tracking) 등이 있다.
(2) 신호 처리: 신호 처리는 주로 오디오 데이터와 관련된 작업을 포함한다. 이러한 작업은 잡음 제거(noise reduction), 에코 취소(echo cancellation), 품질 개선(quality enhancement) 등을 포함할 수 있다.
(3) 압축: 앞서 언급했듯이 멀티미디어 데이터는 종종 매우 크기 때문에 효율적인 저장과 전송을 위해 압축이 필요하다. 비디오와 오디오 데이터 모두 다양한 압축 방법과 코덱(codec)이 사용된다.
(4) 변환: 특정 형식의 멀티미디어 데이터를 다른 형식으로 변환하는 것도 중요할 수 있다. 예를 들어, RAW 혹은 BMP 형식의 이미지를 JPEG나 PNG로 변환하거나 WAV 형식의 오디오 파일을 MP3로 변환하는 것이다.
(5) 스트리밍: 실시간으로 멀티미디어 컨텐츠를 전송하고 재생하는 기능도 중요하다.

**4) 멀티미디어 통신**

▶ 멀티미디어 통신은 멀티미디어 데이터(텍스트, 오디오, 비디오 등)를 네트워크를 통해 전송하는 과정을 말한다. 이는 일반적인 데이터 통신과는 다른 몇 가지 중요한 요구 사항이 있다.
(1) 대역폭: 멀티미디어 데이터는 대체로 크기가 크므로 높은 대역폭이 필요하다. 이는 특히 고화질 비디오 스트리밍에서 중요하다.
(2) 실시간성: 오디오 및 비디오 데이터의 경우 실시간으로 전송되고 재생되어야 하므로 낮은 지연 시간이 필요하다.
(3) 동기화: 멀티미디어 세션에서 여러 종류의 미디어(예: 오디오와 비디오)가 함께 전송될 때, 이들이 동기화되야 한다.
(4) 에러 처리: 네트워크 에러에 대한 적절한 처리도 중요하다. 일부 에러는 사용자에게 미치는 영향을 최소화하기 위해 복구될 수 있지만, 심각한 에러의 경우 세션을 재시작하거나 다른 조치를 취해야 할 수도 있다.
▶ 멀티미디어 통신을 지원하기 위한 여러 가지 프로토콜과 기술이 개발되었다.
(1) 스트리밍 프로토콜:
RTSP(Real-Time Streaming Protocol), RTP(Real-time Transport Protocol), RTCP(Real-Time Control Protocol) 등의 프로토콜들은 실시간 멀티미다 스트리밍을 지원한다.

(2) VoIP(Voice over IP): VoIP 기술은 인터넷 연결을 사용하여 음성 통신 서비스를 제공하는데 사용된다.
(3) WebRTC(Web Real-Time Communication): WebRTC는 웹 브라우저 간에 실시간 음성, 비де오, 데이터 공유를 가능하게 하는 오픈 소스 프로젝트이다.
(4) CDN(Content Delivery Network): CDN은 사용자에게 최적의 경로와 위치에서 컨텐츠를 제공하여 전송 속도를 빠르게 하고 지연 시간을 줄인다.
(5) MPEG-DASH, HLS 등의 적응형 스트리밍 프로토콜: 이들은 네트워크 상황에 따라 동적으로 비디오 품질을 조정하며, 이를 통해 사용자 경험을 최적화한다.

**5) 편집도구 및 저작도구**

▶ 멀티미디어 프로그래밍에서 편집 도구와 저작 도구는 사용자가 멀티미디어 콘텐츠를 생성, 수정 및 조합하는 데 필요한 소프트웨어이다.
▶ 편집 도구: 편집 도구는 이미지, 오디오, 비디오 등의 멀티미디어 데이터를 수정하고 가공하는 데 사용된다. 이러한 도구들은 필터링, 크롭, 회전, 리사이즈 등 기본적인 기능부터 더 복잡한 작업까지 수행할 수 있다.
(1) 이미지 편집: Adobe Photoshop, GIMP 등
(2) 오디오 편집: Audacity, Adobe Audition 등
(3) 비디오 편집: Adobe Premiere Pro, Final Cut Pro 등
▶ 저작 도구: 저작 도구는 여러 종류의 멀티미디어 요소(텍스트, 그래픽스, 사운드, 비디오 등)를 결합하여 새로운 멀티미디어 프로젝트를 생성하는데 사용된다. 이러한 도구들은 일반적으로 시간선 기반의 인터페이스를 제공하며 애니메이션, 인터랙션 및 스크립팅 기능을 지원한다.
(1) 웹과 앱 개발: HTML5/CSS3/JavaScript와 같은 웹 기술은 다양한 멀티미디어 요소를 결합하여 대화형 웹 페이지나 앱을 만드는데 사용된다.
(2) 애니메이션과 게임 개발: Unity3D나 Unreal Engine 같은 게임 엔진들은 복잡한 3D 그래픽스와 사운드를 조합하여 게임을 만드는데 사용된다. Adobe Animate 같은 소프트웨어도 2D 애니메이션에 유용하다.

# < 개발 방법론 >

## 1. 개발프로세스

### 1) 기본원리

▶ 개발 프로세스는 소프트웨어나 시스템을 개발하는 데 필요한 일련의 단계와 활동을 의미한다. 이 프로세스는 일반적으로 요구사항 수집, 설계, 구현, 테스팅, 배포 및 유지보수 단계를 포함한다.

(1) 요구사항 정의: 프로젝트의 목표를 이해하고, 사용자의 요구사항을 분석하고 정의하는 단계다. 이 단계에서는 시스템의 기능, 성능, 인터페이스, 제약사항 등을 명확히 정의한다.

(2) 시스템 설계: 요구사항을 바탕으로 시스템의 전체적인 구조와 각 구성 요소의 동작 방식을 설계한다. 이 단계에서는 데이터 모델, 아키텍처, 인터페이스, 알고리즘 등을 정의한다.

(3) 구현: 설계된 시스템을 프로그래밍 언어를 사용해 코드로 구현하는 단계다. 이 단계에서는 코드의 효율성, 가독성, 재사용성 등을 고려해야 한다.

(4) 테스팅: 구현된 시스템이 요구사항을 정확히 만족하는지 검증하는 단계다. 단위 테스트, 통합 테스트, 시스템 테스트, 사용자 인수 테스트 등 다양한 테스트 방법이 있다.

(5) 유지보수: 시스템이 실제 운영 환경에서 안정적으로 동작하도록 유지보수하는 단계다. 이 단계에서는 버그 수정, 기능 추가, 성능 향상, 환경 변화 대응 등의 작업을 수행한다.

▶ CASE(Computer-Aided Software Engineering): 컴퓨터를 활용한 소프트웨어 공학을 의미한다. 이는 소프트웨어 개발 과정에서 요구 분석, 설계, 구현, 테스트, 유지 보수 등의 과정을 자동화하거나 반자동화하는데 사용되는 도구나 기술을 지칭하기도 한다.

CASE 도구는 개발 팀이 코드를 더 효율적으로 작성하고, 문서화하고, 테스트하고, 유지 관리할 수 있도록 지원한다. 이들 도구는 일반적으로 다음과 같은 기능을 제공한다.

(1) 요구사항 관리: 사용자의 요구사항을 추적하고 관리하는 도구이다. 이를 통해 요구사항이 변경되어도 이를 쉽게 관리할 수 있다.

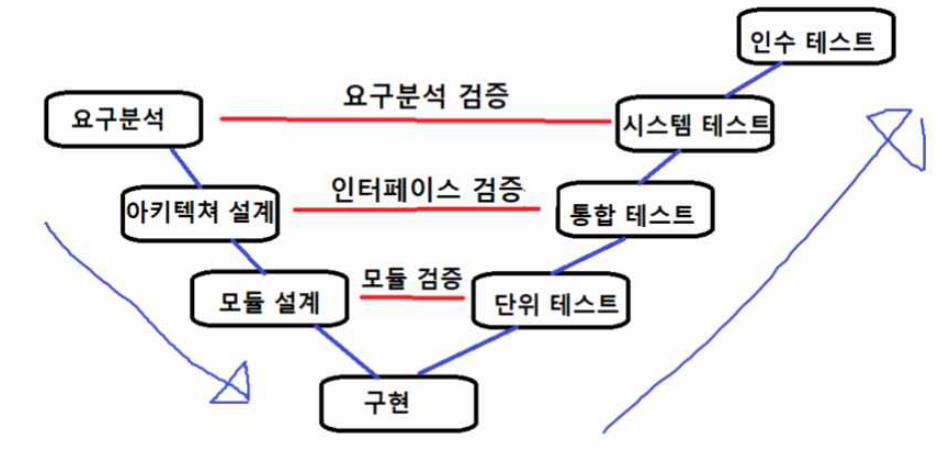
(2) 설계 지원: UML(Unified Modeling Language) 같은 표준화된 언어를 사용하여 소프트웨어의 구조와 동작을 시각적으로 표현할 수 있게 돕는 도구이다.

(3) 코드 생성과 테스트: 개발자가 직접 코드를 작성하는 것을 돕거나, 일부 코드를 자동으로 생성하는 도구이다. 또한, 자동화된 테스트 도구를 통해 코드의 정확성을 검증하는 데 도움을 준다.
(4) 유지 관리와 변경 관리: 이미 배포된 소프트웨어의 유지 관리를 돕거나, 소프트웨어의 버전을 관리하는 도구이다.

**2) 프로세스 모델**

▶ 소프트웨어 개발 프로세스 모델은 소프트웨어를 개발하는 방법론을 의미한다. 각 모델은 특정한 방식으로 요구 사항 분석, 설계, 구현, 테스트 및 유지 보수의 단계를 관리하며, 프로젝트의 목표와 조건에 따라 적합한 모델을 선택한다.
(1) 주먹구구식 모델(Build and fix): 주먹구구식 모델(build and fix)은 공식적인 가이드라인이나 프로세스 없이하는 개발 방식이다. 이 방식은 즉흥적 개발, 코딩과 수정 모델이라고 부르기도 한다. 주먹구구식 모델은 코딩(구현)을 먼저 한 후에 요구분석이나 설계, 유지보수에 대해 생각하는 방식이다. 이 방식은 관리나 유지보수가 굉장히 어렵고, 일을 나눠서 할 수 없으며 프로젝트 진척사항도 알기 어렵다는 문제점이 있다. 따라서 개발자 한 명이 단기간에 작업을 마칠 수 있는 간단한 개발(대학 수업용 프로젝트)에서나 사용되는 방식이다.
(2) 선형 순차적 모델(Linear sequential) 또는 폭포수 모델(Waterfall): 초기에 개발된 전통적인 모델로 표준 프로세스를 정하고 소프트웨어를 순차적으로 개발하는 모델이다. 고전적 생명주기라고도 한다. 폭포수 모델은 계획, 요구분석, 설계, 구현, 테스트, 유지보수의 단계들이 하향식으로 진행되며, 병행이나 거슬러 올라가지 않는다. 따라서 각 단계의 종료마다 확실하게 작업을 종료하고 그 결과를 확인한 뒤 다음 단계로 내려간다.
- 장점
· 관리가 용이하다.
· 체계적으로 문서화할 수 있다.
· 요구사항의 변화가 적은 프로젝트에서 효과적이다.
- 단점
· 각 단계는 이전 단계가 완료되어야 수행된다.
· 각 단계의 결과물이 완벽한 수준이 되어야 다음 단계에서 오류없이 작업을 수행할 수 있다.
· 사용자가 중간에 가시적인 결과를 볼 수 없다.
(3) V모델: V 모델은 폭포수 모델의 변형 모델로, 테스트 단계를 확장한 모델이다. 폭포수 모델이 산출물(문서화) 중점이었다면 V 모델은 각 개발 단계에 대한 검증에

초점을 두어 오류를 줄이는데 중점을 두고 있다.




< V모델 >

(4) 진화적 프로세스 모델(Evolutionary process) 또는 프로토타입 모델(Prototype): 폭포수 모델의 단점이었던 거슬러 올라가는 것이 불가능함에서 오는 요구사항 변화 대처에 약하다라는 문제점에 대응하기 위해 등장했다. 소프트웨어 개발에서 프로토타입은 완전한 소프트웨어 개발 이전에 사용자의 요구에 맞춰 모형 소프트웨어를 만들고 사용자와 의사소통 도구로 이용하게 된다. 프로토타입을 만들고 사용자에게 보여주고, 사용자는 프로토타입을 경험한 뒤 추가적인 요구사항을 준다. 그리고 요구사항이 더 이상 나오지 않을 때까지 반복한다. 이때 프로토타입은 사용자 인터페이스를 중심으로 설계하게 된다.

- 장점

· 프로토타입이 사용자와 개발자간의 의사소통 도구로 사용된다.

· 프로토타입 개발을 반복하기에 사용자의 요구가 충분히 반영된 요구분석명세서가 나온다.

· 초기 프로토타입에서 새로운 요구사항을 발견할 수 있다.

· 사용자의 요구가 충분히 반영되어 유지보수 비용을 줄일 수 있다.

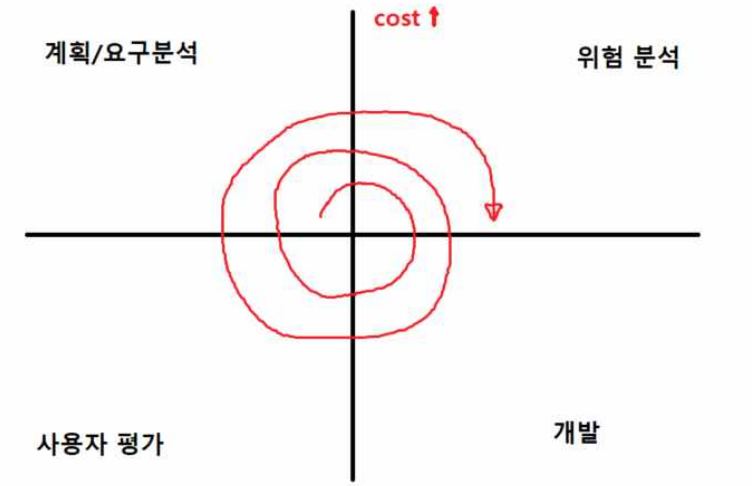
- 단점

· 반복적인 프로토타입 개발로 인해 필요한 인력/비용 산정이 어렵다.

· 프로토타입 개발 과정의 관리/통제가 어렵다.

· 개발 범위가 불명확해 개발 목표나 종료 시점이 불명확해진다.

(5) 나선형 모델: 나선형 모델은 프로토타입 모델에 위험 분석단계가 추가된 모델이다. 위험 분석 단계에서 분석하는 위험은 빈번하게 변경되는 요구사항, 개발 팀원의 경험 부족/팀워크/이직/프로젝트 관리의 부재와 같은 개발 과정에서 어려움을 주는 요소들을 의미한다.




< 나선형 모델 >

- 장점

· 갑작스럽게 발생하는 위험으로 인해 프로젝트가 중단될 확률이 낮아진다.

· 사용자 요구가 충분히 반영된 결과물이 나오기에 사용자의 불만이 적어진다.

- 단점

· 프로젝트 기간이 길어질 수 있다.

· 반복 횟수가 늘어나면 프로젝트 관리가 어려워진다.

· 위험 관리가 중요하기에 위험 관리 전문가가 필요하다.

(6) 단계적 개발 모델(Phased development): 단계적 개발 모델(phased development)은 개발과 사용을 병행하며 개발을 완료해나가는 모델이다. 보통 릴리스 하나를 개발하고, 사용자가 릴리스를 사용할 동안 다음 버전 릴리스를 개발하는 과정을 반복한다. 단계적 개발 모델은 릴리스 구성 방식에 따라 점증적 개발과 반복적 개발로 나뉜다.

- 점증적 개발: 중요하다고 생각되는 부분부터 개발한 뒤, 그 일부를 사용하면서 개발 범위를 늘려나가는 방식이다.

예를 들어 홈페이지를 개발한다고 가정하면, 메인 페이지를 만들어서 사용하게 한다. 그 후 필요에 따라서 기타 기능들을 개발해서 추가시키는 방식이다.

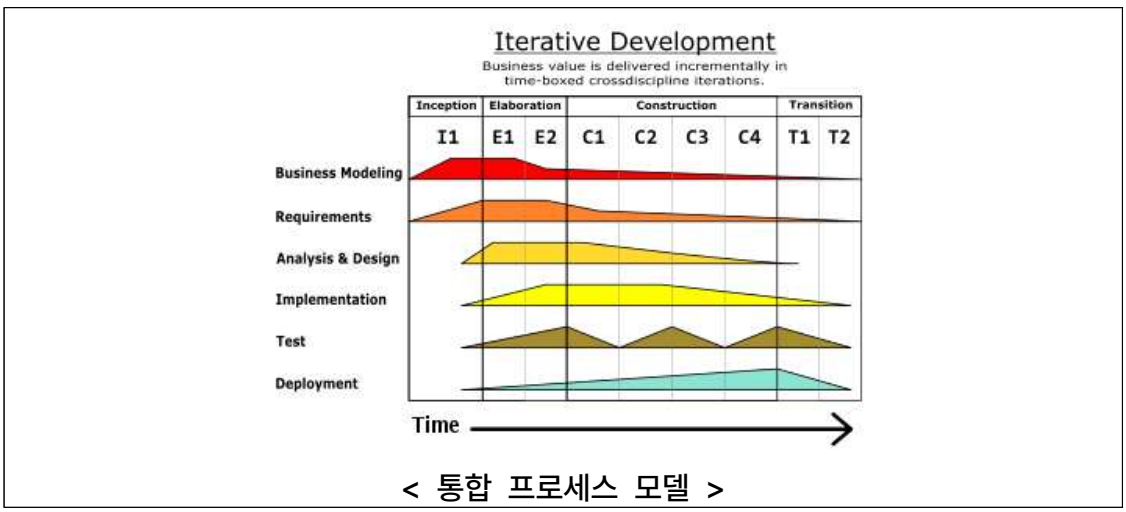
이 방식은 한 번에 많은 비용이 들지 않는다. 또한 완전히 새로운 시스템 전체를 한 번에 줄 때도 조직이 받는 충격을 완화시킬 수 있다. 하지만 서브시스템들이 서로 관련이 있기에 처음에 설계할 때부터 다른 서브시스템과의 연관성을 고려하고 설계해야 한다. 이 과정에서 통합에 어려움을 겪을 수 있다.

- 반복적 개발: 처음에 시스템 전체를 한 번에 개발한다. 그 후 각 서브시스템의 기능과 성능을 변경하거나 보강해서 완성도를 높인다. 이렇게 개발된 릴리스 버전을

내놓으며 개발을 한다. 실제 개발 환경에서는 점증적 개발과 반복적 개발을 함께 사용하는 경우가 많다.

(7) 통합 프로세스 모델(Unified Process): 갈수록 소프트웨어는 규모도 커지고 복잡해지고 있다. 이런 환경에서 요구사항 등의 변화에 유연하게 대처할 수 있도록 개발 과정을 반복해서 사용하는데, 이러한 과정을 반복적 생명주기라고 한다. 대표적으로 반복적 생명주기를 이용하는 모델이 통합 프로세스 모델(Unified Process)이다. 통합 프로세스 모델은 UML(Unified Modeling Language)라는 것과 함께 사용된다. 통합 프로세스 모델은 크게 도입, 구체화, 구축, 전이의 4단계로 구성된다. 이때 각 단계를 여러 개의 작은 단위로 나누어 반복하고 정복해 나가며 진행한다. 각 반복 주기에선 실행 가능한 산출물이 나오고 이를 통해 위험을 관리하게 된다.

- 도입: 비지니스 모델링이나 요구사항 분석이 주로 이루어지는 단계이다.
- 구체화: 분석과 설계가 가장 활발하게 이루어진다.
- 구축: 구현이 가장 활발하게 일어나는 단계이다. 또한 구현에 따른 테스트도 빈번하게 수행된다.
- 전이: 사용자를 위한 제품을 완성시키는 단계이다. 사용자에게 넘기기 위해 배치 작업이 가장 많이 일어난다.




< 통합 프로세스 모델 >

(8) 애자일 프로세스 모델(Agile Process): 고객의 요구사항에 대해 민첩하게 대응하고, 그때 그때 주어지는 문제를 풀어나가는 모델이다. 이 모델은 빈번하게 발생되는 요구사항에 대처하기 위해 변화를 수용하기 쉽도록하는 방법론을 사용한다.
애자일에서 사용하는 방법론에는 다음과 같은 것들이 있다.

- 스크럼, 익스트림 프로그래밍, 적응형 소프트웨어 개발, 린 소프트웨어 개발, 크리스탈 패밀리, 기능 주도 개발론, 동적 시스템 개발 방법론, 애자일 UP
- 애자일 선언문

· 프로세스와 도구 중심이 아닌, 개개인과의 상호 소통을 중시한다.
· 문서 중심이 아닌, 실행 가능한 소프트웨어를 중시한다.
· 계약과 협상 중심이 아닌, 고객과의 협력을 중시한다.
· 계획 중심이 아닌, 변화에 민첩한 대응을 중시한다.
· 즉, 애자일은 고객과의 협업, 단시간에 고객이 작동시킬 수 있는 소프트웨어, 환경과 요구사항 변화에 능동적으로 대처하는 것에 중점을 두고 있다.
- 애자일 프로세스 모델은 반복적인 개발을 통해 자주 출시하는 것을 목표로 하고 있다. 실행가능한 프로토타입을 통해 고객에게 확인을 받고, 단기간에 사용가능한 소프트웨어를 만드는 것을 중요하게 생각한다.
(9) 스크럼(Scrum): 럭비에서 온 단어이다. 간단하게 설명하자면 서로 어깨동무로 팔을 잡은 뒤 하나의 대형으로 집단을 만들어 상대팀을 밀어내는 진형이다. 즉, 팀이 하나가 되어 단단하게 앞으로 나아가는 대형이다.
- 소프트웨어 개발에서 스크럼은 개발 자체보다는 팀의 개선, 프로젝트 관리에 중점을 둔 애자일 방법론이다.
- 스크럼은 구체적인 프로세스를 명확하게 제시하지 않는다. 스크럼은 제품 기능 목록 작성 → 스프린트 계획 회의 → 스프린트 수행 → 스프린트 개발 완료 → 스프린트 완료의 과정을 반복하며 진행된다.
- 제품 기능 목록은 사용자의 요구사항이 적힌 문서이다. 하지만 요구사항에 우선순위가 적용되어 있다는 것이 특징이다.
- 스프린트(sprint)는 전력 질주라는 의미인데, 일반적으로 단거리 달리기 경주(100m)를 의미한다. 개발에서 스프린트는 작업량이 많지 않고 개발 기간도 짧은 것을 의미한다. 즉, 작은 단위의 개발 업무를 단기간에 수행한다는 것을 의미한다. (보통 1달 이내)
- 스프린트를 진행하는 동안 매일 짧고 간단하게 진척사항을 회의한다. (일일 스프린트 회의)
- 스프린트가 종료되면 스프린트 검토 회의와 회고를 하는데, 검토 회의는 실행가능한 산출물을 검토한다. 주로 비즈니스 가치를 점검한다. 그리고 검토 회고는 개선할 점이나 팀 규칙 준수 등을 검토한다. 즉, 회의는 산출물의 품질을 회고는 팀과 팀원에 대한 것을 확인한다.
- 장점
· 반복 주기마다 생성되는 제품으로 사용자와의 의견을 충분히 나눌 수 있다.
· 일일 회의를 통해 팀원 간의 신속한 협조와 조율이 가능하다.
· 자신의 일정을 직접 발표해, 업무 집중 환경이 조성된다.
· 다른 개발 방법론에 비해 단순하고 실천하기 쉽다.

· 프로젝트의 진행상황을 볼 수 있어서 신속하게 목표와 결과의 추정이 쉽다.
- 단점
· 반복 주기마다 실행/테스트 가능한 산춘물이 만들어져야 한다.
· 일일 스크럼 회의가 꼭 짧게 끝날 것을 보장하지 못한다.
· 작업이 얼마나 효율적으로 수행되고 있는지 알기 어렵다.
· 프로젝트 관리에 중점을 두었기 때문에 프로세프 품질을 측정하기 어렵다.

▶ COMET(Concurrent Object Modeling and architectural design mEThod)

: 하드웨어 소프트웨어 동시 진행 시 피드백이 불가능하여 많은 위험요소를 내포하고 있는데, 이러한 위험요소를 회피하기 위해 고안된 모델이며, UML기반 객체 지향 소프트웨어 개발을 위한 방법론 중 하나이다. 이 방법론은 소프트웨어 개발의 초기 단계부터 사용되며, 요구사항 분석, 시스템 설계, 구현 등의 과정에서 사용된다. CASE 도구와 같은 소프트웨어 개발 도구를 사용하여 개발 과정을 지원한다.

- 주요 특징

(1) 병행 모델링: COMET는 시스템의 다양한 측면을 동시에 모델링하는 것을 강조한다. 예를 들어, 기능적 요구사항, 비기능적 요구사항, 사용자 인터페이스, 데이터 모델 등을 동시에 고려한다.

(2) 객체 지향 설계: COMET는 객체 지향 원칙과 패턴을 기반으로 시스템을 설계한다. 이를 통해 시스템의 재사용성, 확장성, 유지 관리성을 향상시킨다.

(3) 아키텍처 중심 접근법: COMET는 시스템의 아키텍처를 중심으로 설계를 수행한다. 이를 통해 시스템의 전체적인 구조와 구성 요소의 상호 작용을 명확하게 이해할 수 있다.

(4) 반복적 개발 프로세스: COMET는 요구사항 분석, 시스템 설계, 구현, 테스트 등의 과정을 반복적으로 수행하는 것을 권장한다. 이를 통해 시스템의 품질을 점진적으로 향상시키고, 변경 사항을 쉽게 수용할 수 있다.

▶ 럼바우 객체 모델링(Rumbaugh's Object Modeling)

: 제임스 럼바우(James Rumbaugh)가 개발한 객체 지향 분석/설계 방법론이다. 이 방법론은 객체 모델링 기법(OMT, Object Modeling Technique)이라고도 불린다. 럼바우의 객체 모델링 기법은 객체 모델, 동적 모델, 기능 모델을 동시에 사용하여 시스템을 분석하고 설계한다. 이 세 가지 모델은 서로 다른 관점에서 시스템을 바라보는 것이며, 이들을 통합하여 시스템의 전체적인 모습을 이해하고 분석하는 것이 가능하다.

(1) 객체 모델링(Object Model): 시스템을 구성하는 객체들과 객체들 간의 관계를 표현한다. 객체 모델에서는 클래스, 인스턴스, 속성, 연산, 관계 등이 정의된다.

(2) 동적 모델링(Dynamic Model): 시스템의 동작 상태와 시간에 따른 변화를 표현한다. 상태-이벤트 다이어그램(State-Event Diagram)을 사용하여 시스템의 동적인 행동을 표현한다.
(3) 기능 모델링(Functional Model): 시스템의 기능적인 측면을 표현한다. 데이터 흐름 다이어그램(Data Flow Diagram)을 사용하여 시스템의 처리 과정과 데이터 흐름을 표현한다.

### 3) 요구사항 분석

▶ 요구사항 분석은 개발 프로세스의 매우 중요한 단계로, 시스템이 수행해야 할 기능과 서비스, 그리고 제약 조건 등을 이해하고 정의하는 과정이다. 이 단계에서는 사용자의 요구사항을 수집하고, 그 요구사항을 명확하고 완전하며 일관된 형태로 문서화하는 작업을 수행한다.
(1) 요구사항 수집: 사용자 인터뷰, 설문조사, 워크숍, 문서 검토 등의 방법을 통해 사용자의 요구사항을 수집한다. 이 단계에서는 사용자가 실제로 무엇을 원하는지 이해하는 것이 중요하다.
(2) 요구사항 분류: 수집된 요구사항을 기능적 요구사항과 비기능적 요구사항으로 분류한다. 기능적 요구사항은 시스템이 수행해야 할 특정 기능을 설명하며, 비기능적 요구사항은 시스템의 성능, 보안, 사용성 등의 품질 속성을 설명한다.
(3) 요구사항 명세: 요구사항을 명확하고 구체적인 언어로 문서화한다. 이 문서는 개발 팀과 사용자가 요구사항에 대해 공통의 이해를 갖도록 돕는다.
(4) 요구사항 검증: 요구사항이 완전하고 일관되며 실행 가능한지 확인한다. 이 단계에서는 문제점을 찾아내고 수정하여 요구사항의 품질을 향상시킨다.
(5) 요구사항 관리: 요구사항은 프로젝트 기간 동안 변할 수 있다. 따라서 요구사항의 변경을 관리하고, 변경된 요구사항을 문서화하고 팀원들에게 전달하는 것이 중요하다.
▶ 다양한 요구사항
(1) 기능적 요구 사항(Functional Requirements): 시스템이 제공해야 하는 기능에 대한 것이다.
(2) 비기능적 요구 사항(Non-functional Requirements): 시스템의 성능, 보안, 사용성 등과 같은 속성에 대한 것이다.
(3) 제약 조건(Constraints): 법률적 제약조건 또는 비즈니스 규칙 등으로 인해 발생하는 제약 조건이다.

### 4) 시스템 아키텍처

▶ 시스템 아키텍처는 소프트웨어 시스템의 전체적인 구조와 그 구성 요소, 그리고 이들 간의 관계를 정의한 것이다. 시스템 아키텍처는 시스템이 어떻게 동작할지, 어떤 기능을 수행할지, 그리고 어떤 제약 사항이 있는지를 설명한다.

▶ 시스템 아키텍처의 요소

(1) 하드웨어 아키텍처: 하드웨어 아키텍처는 시스템에 사용되는 물리적 장치들과 이들 간의 연결을 설명한다. 예를 들어, 서버와 클라이언트 컴퓨터, 네트워크 장비 등과 같은 것들이 여기에 포함된다.

(2) 소프트웨어 아키텍처: 소프트웨어 아키텍처는 시스템에서 실행되는 프로그램 및 애플리케이션의 구조와 동작 방식을 설명한다. 예를 들어, 데이터베이스 서버, 웹 서버, 애플리케이션 서버 등과 같은 것들이 여기에 포함된다.

▶ 시스템 아키텍처 설계 과정

(1) 요구사항 분석: 시스템이 충족해야 할 기능적 요구사항과 비기능적 요구사항(성능, 보안, 확장성 등)을 분석한다.

(2) 시스템 분해: 시스템을 더 작고 관리하기 쉬운 구성 요소나 모듈로 분해한다. 이 단계에서는 각 구성 요소의 역할과 책임을 정의한다.

(3) 구성 요소 배치: 각 구성 요소가 어떻게 배치되고, 서로 어떻게 상호작용할지 결정한다. 이 단계에서는 데이터 흐름, 컨트롤 흐름, 의존성 등을 고려한다.

(4) 아키텍처 스타일 결정: 적절한 아키텍처 스타일(레이어드, 클라이언트-서버, 마이크로서비스 등)을 선택한다. 아키텍처 스타일은 시스템의 기본 구조와 통신 패턴을 결정한다.

(5) 아키텍처 검증: 설계된 아키텍처가 요구사항을 만족하고, 시스템의 품질 속성을 충족하는지 검증한다.

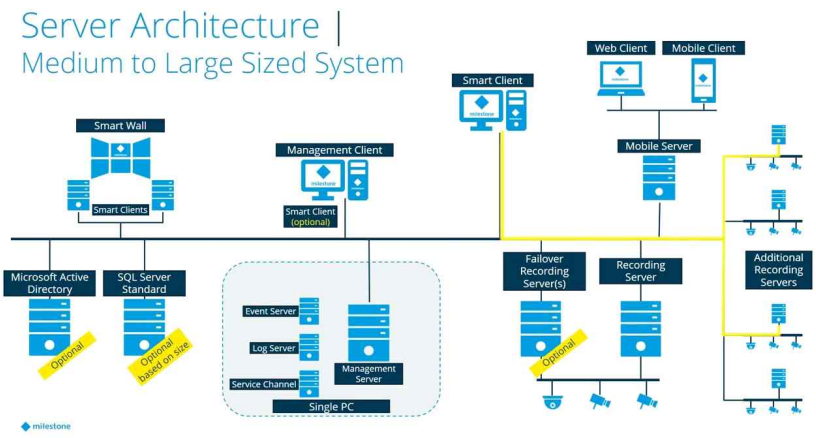
▶ 시스템 아키텍처 디자인하는 과정에서 다음과 같은 사항들을 고려해야 한다.

(1) 성능: 시스템이 사용자 요구사항을 충족하기 위해 필요한 성능 수준을 제공해야 한다.

(2) 신뢰성: 시스템이 안정적으로 작동하고 오류 없이 기능을 수호하는 것은 매우 중요하다.

(3) 보안: 데이터 보호 및 프라이버시 보장 등 보안 요구사항도 고려되어야 한다.

(4) 유지 관리 및 확장성: 시스템이 쉽게 업그레이드되거나 확장될 수 있도록 설계하는 것도 중요하다.




< 시스템 아키텍처 예시 >

### 5) 설계기법

▶ 소프트웨어 설계는 소프트웨어 개발 프로세스에서 중요한 단계로, 시스템의 구조, 구성 요소, 인터페이스, 동작 방식 등을 정의하는 역할을 한다. 이는 개발자가 코드를 작성하기 전에 시스템의 전반적인 틀을 이해하고 계획하는 데 도움을 준다.

(1) 모절차적 설계: 최초의 소프트웨어 설계 방법론으로, 시스템을 순차적이고 계층적인 프로세스로 표현한다. 각 프로세스는 하위 프로세스로 분해될 수 있으며, 이는 코드의 구조화와 모듈화를 쉽게 만든다.

(2) 객체 지향 설계: 시스템을 상호작용하는 객체들의 집합으로 보는 방법론이다. 각 객체는 자신만의 상태와 행동을 가지며, 다른 객체와 메시지를 주고받아 시스템의 기능을 수행한다.

(3) 함수형 설계: 시스템을 순수 함수의 조합으로 보는 방법론이다. 이는 상태 변경이나 부작용을 최소화하며, 이는 코드의 예측 가능성과 재사용성을 향상시킨다.

(4) 서비스 지향 설계: 시스템을 독립적인 서비스들의 집합으로 보는 방법론이다. 각 서비스는 특정 비즈니스 기능을 수행하며, 다른 서비스와 통신하여 복잡한 작업을 수행한다.

(5) 컴포넌트 기반 설계: 시스템을 재사용 가능한 컴포넌트들의 집합으로 보는 방법론이다. 각 컴포넌트는 독립적으로 배포하고 사용할 수 있으며, 이는 시스템의 확장성과 유지 관리성을 향상시킨다.

### 6) 소프트웨어 테스팅

▶ 소프트웨어 테스팅은 개발된 소프트웨어가 기대하는 동작을 정확하게 수행하는지, 그리고 요구사항을 충족하는지 확인하는 과정이다. 이는 소프트웨어의 품질을

보장하고, 버그나 오류를 찾아내어 수정하는 데 필요한 작업이다.

▶ 소프트웨어 테스팅 종류

(1) 정적 테스트(Static Testing): 코드나 문서를 직접 실행하지 않고 검토하는 테스팅 방식이다. 코드 리뷰, 피어 리뷰, 정적 분석 도구의 사용 등이 포함된다.

(2) 동적 테스트(Dynamic Testing): 프로그램을 실제로 실행하며 오류를 찾는 테스팅 방식이다. 단위 테스트, 통합 테스트, 시스템 테스트 등이 이 방식에 속한다.

(3) 블랙박스 테스트(Black-Box Testing): 이 방법은 소프트웨어의 내부 구조나 작동 원리를 고려하지 않고 테스트를 수행한다. 블랙박스 테스트는 주로 입력에 대한 출력이 예상대로 동작하는지 검사하는데 초점을 맞추는 방법이며 특정 기능을 알기 위해서 각 기능이 완전히 작동되는 것을 입증하는 테스트로, '기능 테스트'라고도 한다. 이 방식은 사용자의 관점에서 소프트웨어를 테스트하며, 테스트 케이스는 일반적으로 기능 명세, 요구사항, 또는 사용 사례를 기반으로 생성된다. 블랙박스 테스트는 소프트웨어가 실제로 사용될 때 발생할 수 있는 다양한 시나리오를 테스트하므로, 소프트웨어의 사용성과 신뢰성을 향상시키는 데 도움이 된다.

- 특징

· 사용자의 요구사항 명세를 보면서 테스트하는 것으로, 주로 구현된 기능을 테스트한다.

· 소프트웨어 인터페이스에서 실시되는 테스트이다.

· 부정확하거나 누락된 기능, 인터페이스 오류, 자료 구조나 외부 데이터베이스 접근에 따른 오류, 행위나 성능 오류, 초기화와 종료 오류 등을 발견하기 위해 사용되며, 테스트 과정의 후반부에 적용된다.

- 종류

· 동치 분할 검사(Equivalence Partitioning Testing): 입력 자료에 초점을 맞춰 테스트 케이스를 만들고 검사하는 방법으로 동등 분할 기법이라고도 한다.

· 경계값 분석(Boundary Value Analysis): 입력 자료에만 치중한 동치 분할 기법을 보완하기 위한 기법이며, 입력 조건의 중간값보다 경계값에서 오류가 발생될 확률이 높다는 점을 이용하여 입력 조건의 경계값을 테스트 케이스로 선정하여 검사하는 기법이다.

· 원인-효과 그래프 검사(Cause-Effect Graphing Testing): 입력 데이터 간의 관계와 출력에 영향을 미치는 상황을 체계적으로 분석한 다음 효용성이 높은 테스트 케이스를 선정하여 검사하는 기법이다.

· 오류 예측 검사(Error Guessing): 과거의 경험이나 확인자의 감각으로 테스트하는 기법으로 다른 블랙 박스 테스트 기법으로는 찾아낼 수 없는 오류를 찾아내는 일력의 보충적 검사 기법, 데이터 확인 검사라고도 한다.

· 비교 검사(Comparison Testing): 여러 버전의 프로그램에 동일한 테스트 자료를 제공하여 동일한 결과가 출력되는지 테스트하는 기법이다.

(4) 화이트박스 테스트(White-Box Testing): 이 방법은 소프트웨어의 내부 구조와 동작을 이해하고 테스트를 수행한다. 화이트박스 테스트는 코드의 특정 섹션, 결정점, 경로 등을 직접 검사한다. 이 방식은 테스트 케이스를 작성하기 위해 소스 코드에 대한 세부적인 지식이 필요하다. 화이트박스 테스트는 코드의 복잡성, 경로, 조건 등을 검사하여 놓칠 수 있는 버그나 결함을 찾아내는 데 효과적이다.

- 특징

· 화이트박스 테스트는 설계된 절차에 초점을 둔 구조적 테스트며, 테스트 과정의 초기에 적용된다.

· 모듈 안의 작동을 직접 관찰한다.

· 원시 코드(모듈)의 모든 문장을 한 번 이상 실행함으로써 수행된다.

· 프로그램의 제어 구조에 따라 선택, 반복 등의 분기점 부분들을 수행함으로써 논리적 경로를 제어한다.

- 종류

· 기초 경로 검사(Base Path Testing): 대표적인 화이트박스 테스트 기법이다. 테스트 케이스 설계자가 절차적 설계의 논리적 복잡성을 측정할 수 있게 해주는 테스트 기법이고, 테스트 측정 결과는 실행 경로의 기초를 정의하는 데 지침으로 사용된다.

· 제어 구조 검사(Control Structure Testing): 프로그램 모듈 내에 있는 논리적 조건을 테스트하는 테스트 케이스 설계 기법인 '조건 검사(Condition Testing)'와 프로그램의 반복(Loop) 구조에 초점을 맞춰 실시하는 테스트 케이스 설계 기법인 '루프 검사(Loop Testing)'와 프로그램에서 변수의 정의와 변수의 사용 위치에 초점을 맞춰 실시하는 테스트 케이스 설계 기법인 '데이터 흐름 검사(Data Flow Tesing)'를 사용한다.

(5) 회귀 테스트(Regression Testing): 수정이나 업데이트 후에도 기존 기능이 올바르게 작동하는지 확인하기 위해 이미 테스트된 프로그램의 부분을 다시 테스트하는 방식이다.

(6) 탐색적 테스트(Exploratory Testing): 테스트 스크립트를 미리 작성하지 않고, 테스터의 경험과 직관에 기반하여 테스트를 설계하고 실행하는 방식이다.

(7) 부하 테스트(Load Testing): 시스템에 과도한 요청이나 부하를 가해 시스템의 성능을 측정하고 한계를 확인하는 방식이다.

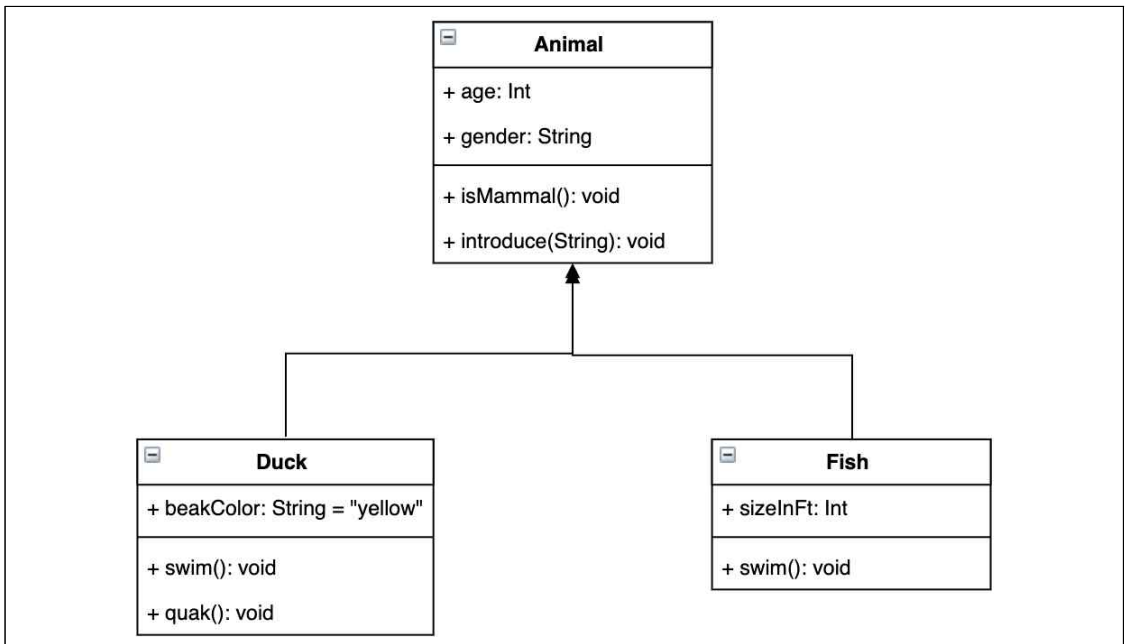
### 7) UML 다이어그램

▶ UML(Unified Modeling Language)은 소프트웨어 시스템을 시각적으로 표현하기 위한 표준 모델링 언어다. UML은 복잡한 시스템을 이해하고 설계하며, 분석하는 데 도움을 주는 다양한 다이어그램을 제공한다.

▶ UML 다이어그램의 주요 종류

(1) 클래스 다이어그램(Class Diagram): 클래스 다이어그램이란 시스템에서 사용되는 객체 타입을 정의한다. 그들 간의 존재하는 정적인 관계를 다양한 방식으로 표현한 다이어그램이다. 객체 지향 시스템 모델링에서 가장 공통적으로 많이 쓰이는 다이어그램이다. 바로 프로그램 코드로 변환이 가능하다.

- 구조

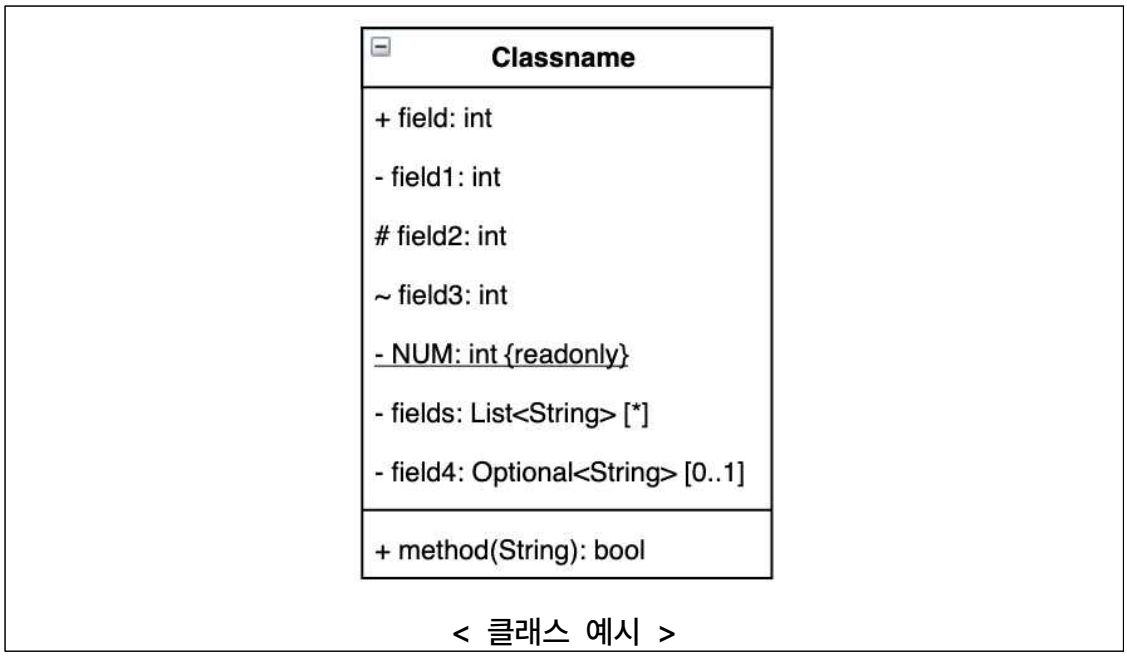



< 클래스 다이어그램 구조 예시 >

: 클래스는 이름(name), 속성(attribute), 연산(operation)으로 구성이 된다. 클래스 박스를 세 부분으로 나누었을 때, 첫 번째 영역에는 이름, 두 번째 부분은 속성, 세 번째 부분은 연산을 기술한다.

- 요소

· 클래스: 클래스 다이어그램에서 클래스는 이름, 속성(변수), 메소드 순으로 나열합니다. 속성과 메소드는 생략이 가능하지만 이름은 필수로 입력해야 한다.

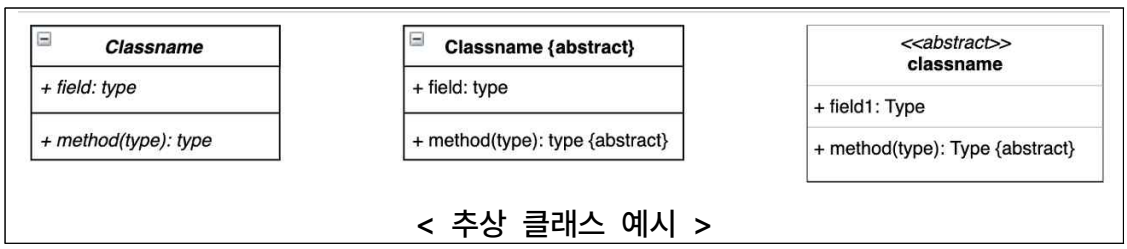



< 클래스 예시 >

· 스테레오 타입: 인터페이스나 추상 클래스와 같은 요소를 표기하기 위해 << >> 와 같은 문법을 사용하는데 이를 길러멧(guillemet)이라 부른다.

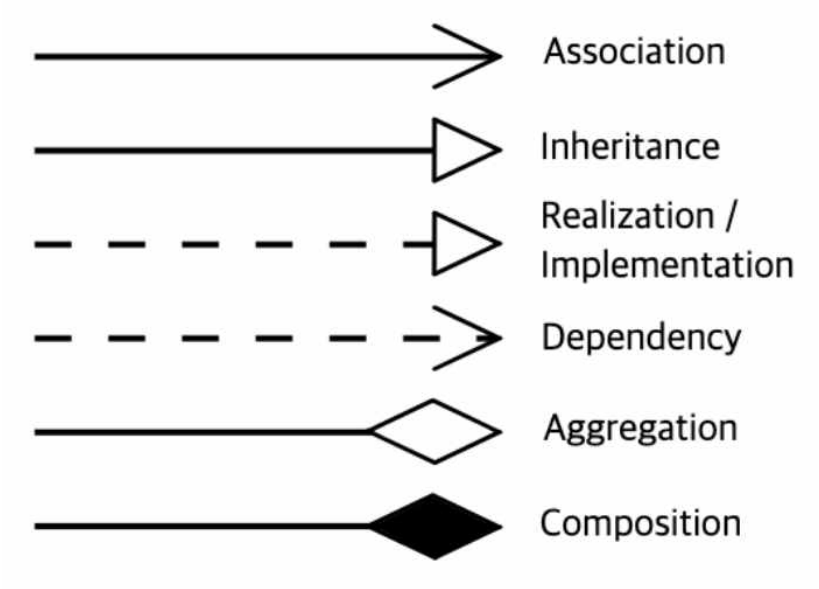

< 스테레오 타입 예시 >

· 추상 클래스: 추상 클래스를 나타내는 방법은 총 3가지로 이탤릭체(기울어진 글씨)로 표시하거나 클래스 명에 {abstract}을 붙이거나 길러멧으로 표시하는 방법이 있다.




< 추상 클래스 예시 >

- 관계




< 클래스 다이어그램 관계 기호 >

· 의존(Dependency): 하나의 모델 요소가 다른 모델 요소를 사용하는 관계

· 연관(Association): 클래스로부터 생성된 인스턴스들 간의 관계

· 집합 연관(Aggregation): 전체와 부분을 나타내는 모델 요소, 전체와 부분은 서로 독립적이다.

· 복합 연관(Composition): 전체와 부분을 나타내는 모델 요소, 연관 관계를 맺고 있는 클래스의 생명주기가 같다.

· 상속(Inheritance): 상속 관계를 나타낸다.

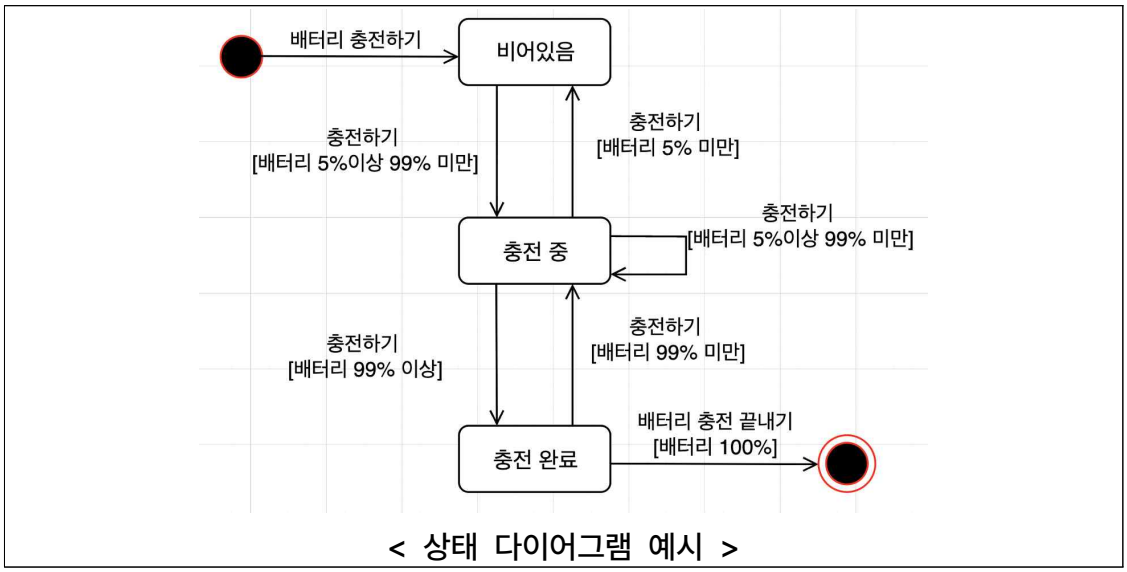
· 실체화(Realization): 인터페이스를 상속하여 실제 기능을 실현화 할 때 사용한다.

(2) 인터랙션 다이어그램(Interaction Diagram): 시스템이 수행하는 과정을 시각화하기 위한 다이어그램이다. Usecase 다이어그램의 actor와 개념 클래스 다이어그램 객체 사이의 상호 작용을 표현한다. UML에는 두 가지 형태의 인터랙션 다이어그램이 있다. 객체 간의 메시지 흐름을 표현하기 위한 시퀀스 다이어그램과 커뮤니케이션 다이어그램이 있다.

- 시퀀스 다이어그램(Sequence Diagram): 문제 해결을 위한 객체를 정의하고 객체간의 상호작용 메시지 시퀀스를 시간의 흐름에 따라 나타내는 다이어그램이다.

- 커뮤니케이션 다이어그램(Communication Diagram): 시스템의 객체들 간에 메시지를 주고받는 상호작용을 표시하는 다이어그램이다. 커뮤니케이션 다이어그램은 시퀀스 다이어그램과 유사한 정보를 제공하지만, 각 객체 간의 관계에 더욱 초점을 맞춘다. 다이어그램은 객체들을 노드로 표현하고, 객체들 간의 관계는 선으로 나타낸다. 이 선 위에는 객체 간에 교환되는 메시지와 그 순서를 표시한다.

(3) 상태 다이어그램(State Diagram): 객체가 가질 수 있는 모든 상태를 표현하며 특정 객체에 대하여 사전 발생에 따른 상태 천이 과정을 묘사한 다이어그램이다. 진입 및 탈출 조건, 상태 전이에 필요한 사건 등 자세한 사항에 대한 표현이 가능하며 설계 단계에서 객체의 동적인 행동을 표현하는데 주로 사용된다.




< 상태 다이어그램 예시 >

(4) 컴포넌트 다이어그램(Component Diagram): 시스템의 물리적인 구조를 나타내며, 시스템을 구성하는 컴포넌트와 그 관계를 표현한다.

(5) 객체 다이어그램(Object Diagram): 시스템의 인스턴스 수준에서 정적 구조를 보여준다. 특정 시점에서 객체간의 관계와 객체의 상태를 나타낸다.

(6) 활동 다이어그램(Activity Diagram): 시스템 내의 작업 흐름이나 비즈니스 프로세스를 표현한다.

## 2. 프로젝트관리

### 1) 프로젝트관리 개요

▶ 프로젝트 관리는 목표를 달성하기 위해 특정한 작업을 계획, 조직, 관리하는 전체 과정을 의미한다. 이는 주어진 시간과 자원 내에서 프로젝트의 목표를 성공적으로 달성하기 위해 필요한 일련의 활동을 포함하며, 다음과 같은 핵심 요소들이 있다.

(1) 목표 설정: 프로젝트의 목적과 목표를 분명하게 정의하고 이해하는 것이 중요하다.

(2) 계획 수립: 프로젝트 계획은 필요한 작업, 일정, 예산 등을 정의한다. 또한 리

스크 관리 계획도 포함되어야 한다.

(3) 자원 할당: 프로젝트에 필요한 자원(인력, 장비, 자재 등)을 적절하게 배분하는 것이 중요하다.

(4) 작업 실행 및 모니터링: 팀이 계획대로 작업을 수행하면서 진척 상황을 지속적으로 모니터링하고 필요에 따라 조정해야 한다.

(5) 위기 및 변화 관리: 예상치 못한 문제가 발생할 경우 적시에 대응하여 해결해야 하며, 변화 요구사항도 유연하게 처리할 수 있어야 한다.

(6) 커뮤니케이션 관리: 팀 내외부와의 커뮤니케이션은 매우 중요하다. 정보 공유와 업데이트는 정확하고 시간적으로 이루어져야 한다.

(7) 클로징(마무리): 프로젝트가 완료되면 결과물을 검증하고 고객에게 전달한다. 그 후 프로젝트 평가 및 회고를 진행하여 앞으로의 개선 사항을 도출할 수 있다.

**2) 품질관리**

▶ 프로젝트 관리에서 품질 관리는 프로젝트 결과물이 고객의 요구사항과 기대치를 만족시키는지 확인하는 과정이다. 이는 품질 계획 수립, 품질 보증, 그리고 품질 제어의 세 가지 주요 활동으로 구성된다.

(1) 품질 계획 수립: 프로젝트 초기에 진행되며, 프로젝트의 목적과 요구사항을 바탕으로 어떤 기준이나 접근법을 사용하여 품질을 측정하고 관리할 것인지를 결정한다.

(2) 품질 보증: 이 활동은 프로젝트 전반에 걸쳐 진행되며, 계획된 품질 접근법이 적절하게 실행되고 있는지를 확인한다. 이를 위해 정기적인 검사와 감사가 수행될 수 있다.

(3) 품질 제어: 이는 결과물이 명시된 품질 기준을 충족하는지 확인하는 과정이다. 이 단계에서 발견된 문제들은 수정 작업 후 재검사가 필요할 수 있다.

**3) C 국제표준(ISO/IEC9899) 개요**

▶ ISO/IEC 9899는 국제 표준화 기구(ISO)와 국제 전기 표준화 위원회(IEC)가 공동으로 제정한 C 언어에 대한 표준이다. 이는 C 프로그래밍 언어의 문법, 데이터 타입, 라이브러리 등을 정의하며, 이를 통해 다양한 하드웨어 및 운영 체제에서 일관된 동작을 보장한다.

▶ C 언어의 최초 표준인 ANSI X3.159-1989 (일명 "ANSI C" 또는 "C89")은 1989년에 미국 국립표준협회(ANSI)에 의해 발행되었다. 이후 ISO와 IEC는 ANSI C를 기반으로 ISO/IEC 9899:1990 (일명 "C90")을 제정하였다.

그 후, 여러 개정판이 발행되었다.

(1) ISO/IEC 9899:1999 ("C99"): 주요 변경 사항으로는 새로운 데이터 타입과 연산자, 인라인 함수, 변수 길이 배열 등이 추가되었다.
(2) ISO/IEC 9899:2011 ("C11"): 멀티스레딩 지원과 안전성 강화를 위한 기능들이 추가되었다.
(3) ISO/IEC 9899:2018 ("C18"): 이 버전은 기존의 C11 표준에 대한 오류 수정 및 명확성 개선을 목적으로 한다. 새로운 기능은 추가되지 않았다.