C++ 강좌 11회 : 중첩된 구조체와 상속

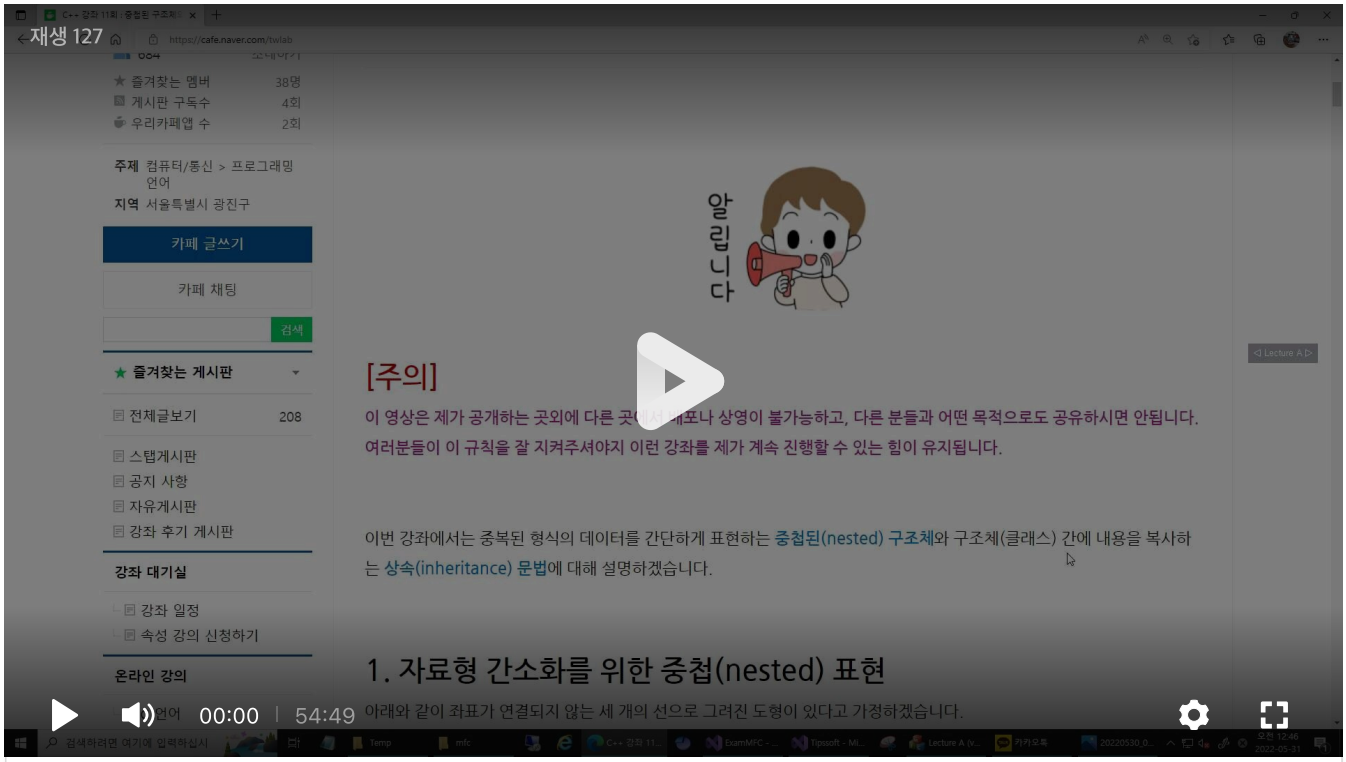
김성엽 카페매니저 + 구독 1:1 채팅

2022.05.29. 04:00 조회 327

[주의]

이 영상은 제가 공개하는 곳외에 다른 곳에서 배포나 상영이 불가능하고, 다른 분들과 어떤 목적으로도 공유하시면 안됩니다. 여러분들이 이 규칙을 잘 지켜주셔야지 이런 강좌를 제가 계속 진행할 수 있는 힘이 유지됩니다.

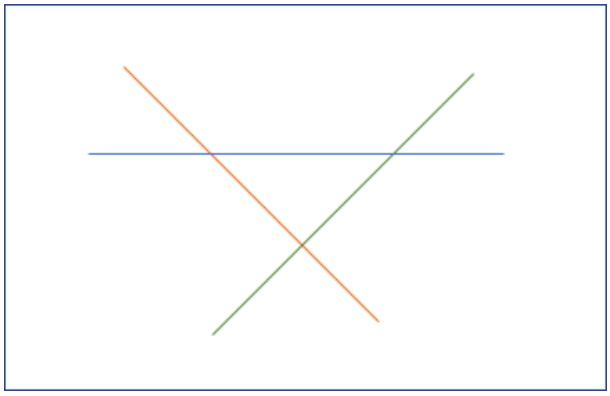
이번 강좌에서는 중복된 형식의 데이터를 간단하게 표현하는 중첩된(nested) 구조체와 구조체(클래스) 간에 내용을 복사하는 상속(inheritance) 문법에 대해 설명하겠습니다.



C++ 강좌 11회 : 중첩된 구조체와 상속

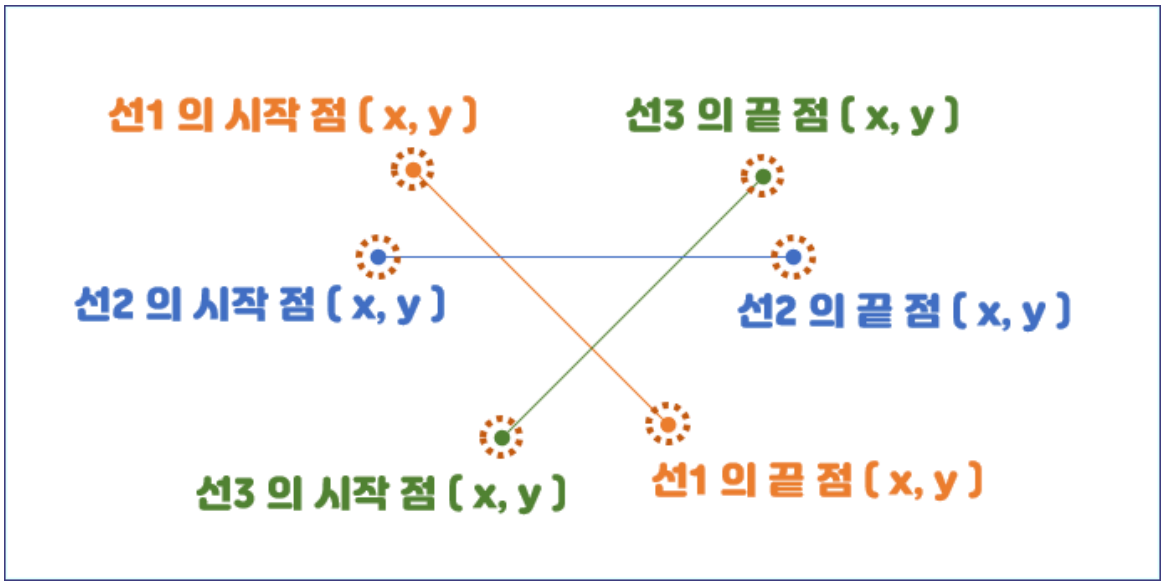
1. 자료형 간소화를 위한 중첩(nested) 표현

아래와 같이 좌표가 연결되지 않는 세 개의 선으로 그려진 도형이 있다고 가정하겠습니다.



이 도형을 구성하는 좌표를 저장하려면 어떻게해야 할까요?

아래의 그림처럼 선을 그리려면 시작 점(x, y)과 끝 점(x, y)으로 정보가 구성되기 때문에 배열 문법을 사용해서 각 선의 좌표를 저장하는 코드를 작성하면 됩니다.




```
int start_x[3], start_y[3]; // 선의 시작점 3개
int end_x[3], end_y[3]; // 선의 끝점 3개
```

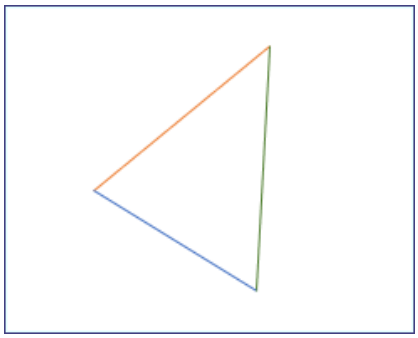
그런데 위 도형이 3개의 선이 아니라 최소 3개에서 최대 9개의 선으로 구성된 수 있다는 조건이 추가되면 이 도형의 좌표를 저장하는 코드는 아래와 같이 변경될 것입니다. 도형을 구성할 선이 몇 개인지 모르기 때문에 구성된 선의 수를 저장할 변수를 추가하고 좌표를 저장할 배열의 크기는 사용 가능한 최댓값으로 변경했습니다.

```
#define MAX_COUNT   9

int main()
{
    int count; // 구성된 선의 수
    int start_x[MAX_COUNT], start_y[MAX_COUNT]; // 선의 시작점
    int end_x[MAX_COUNT], end_y[MAX_COUNT]; // 선의 끝점

    return 0;
}
```

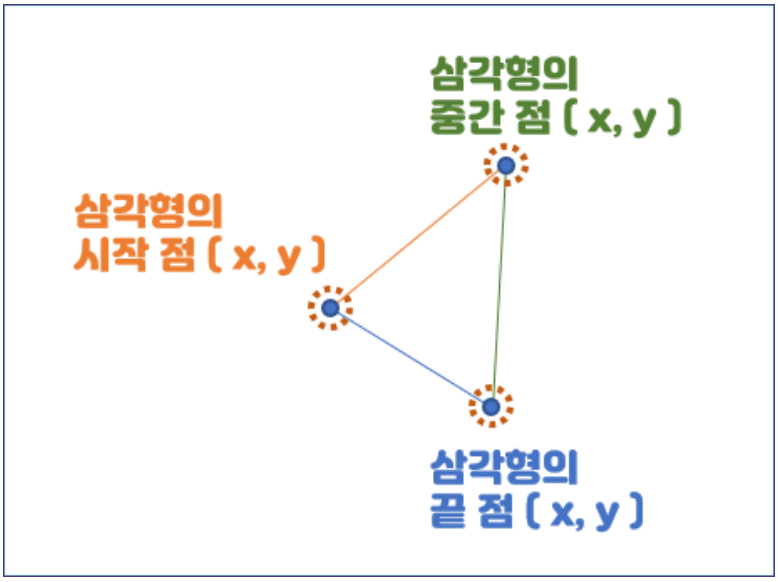
그런데 count, start_x, start_y, end_x, end_y 변수는 **같은 대상(도형)을 구성하는 값이기 때문에 이 데이터들을 그룹지어서 표현**하는 것이 더 좋습니다. 그래서 구조체를 사용해서 아래와 같이 이 데이터를 하나의 자료형으로 정의하고 변수를 선언하는 방식으로 코드를 수정했습니다.

```
#define MAX_COUNT   9

// 여러 개의 선으로 구성된 도형의 좌표를 기억할 자료형
struct MultiLineData
{
    int count;  // 선의 개수
    int start_x[MAX_COUNT], start_y[MAX_COUNT]; // 선의 시작점
    int end_x[MAX_COUNT], end_y[MAX_COUNT]; // 선의 끝점
};

int main()
{
    MultiLineData data;

    // 첫 번째 선의 시작 좌표를 (5, 12)로 끝 좌표를 (120, 200)으로 설정
    data.count = 1;
    data.start_x[0] = 5;
    data.start_y[0] = 12;
    data.end_x[0] = 120;
    data.end_y[0] = 200;

    return 0;
}
```

이번에는 아래의 그림처럼 삼각형 모양의 도형을 저장해 봅시다.



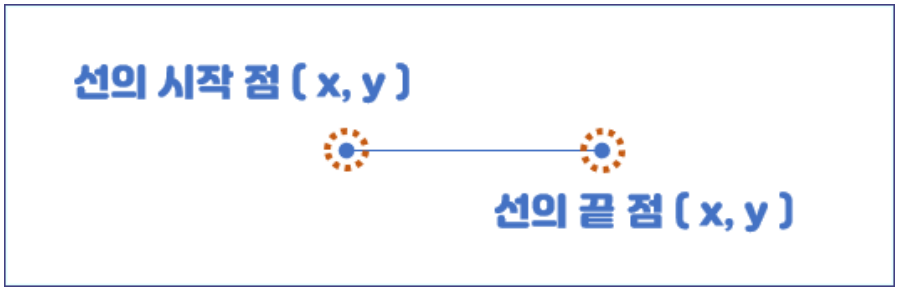
이 도형은 이전 도형처럼 **세 개의 선으로 이루어져있지만 선이 모두 연결**되어 있기 때문에 아래의 그림처럼 여섯 개의 좌표가 아닌 **세 개의 좌표만 저장**하면 됩니다. 그리고 삼각형이기 때문에 점의 개수도 바뀌지 않습니다.



따라서 위 도형을 저장하는 구조체는 아래와 같이 단순하게 정의할 수 있습니다.

```
// 연속 선을 구성하는 좌표를 기억할 자료형
struct PolyLineData
{
    int x[3], y[3]; // 삼각형을 구성하는 세 개의 좌표
};
```

마지막으로 아래와 같이 선 모양을 저장해 봅시다. 선을 저장하는 구조체는 **두 개의 점은 연결**한 것이기 때문에 **두 개의 좌표 값만 저장**하면 됩니다.



```
// 선을 구성하는 좌표를 기억할 자료형
struct LineData
{
    int x[2], y[2]; // 선을 구성하는 두 개의 좌표
};
```

이렇게 도형이 추가될 때마다 해당 도형의 특성을 분석해서 대응하다보면 구조체의 개수는 계속 증가할 것입니다. 프로그래밍에서 구조체는 사용자 정의 자료형이기 때문에 **구조체의 개수가 계속 증가한다는 것은 자료형의 개수가 계속 증가한다는 뜻**입니다. 그리고 **자료형이 많아지면 자료형을 구분하여 처리하기 위해 자연스럽게 조건문이 많아지고** 자료형 변화에 따른 대처를 위해 함수도 많아지게 됩니다. 함수가 많아진다는 것이 단점은 아니지만 자료형이 다르다는 이유로 비슷한 기능의 함수를 다시 만드는 것은 효율적이지 않습니다.

따라서 개발자는 이렇게 자료형의 개수가 증가하기 시작하면 **공통점은 파악해서 자료형의 개수를 줄이거나 중첩된(nested) 표현으로 자료형 간에 연관성은 찾아 조건문이나 함수가 많아지는 문제를 해결**해야 합니다. 그렇다면 지금까지 우리가 도형을 저장하기 위해 작성한 구조체를 한 번 살펴봅시다.

```
#define MAX_COUNT   9

// 여러 개의 선으로 구성된 도형의 좌표를 기억할 자료형
struct MultiLineData
{
    int count;  // 선의 개수
    int start_x[MAX_COUNT], start_y[MAX_COUNT]; // 선의 시작점
    int end_x[MAX_COUNT], end_y[MAX_COUNT]; // 선의 끝점
};

// 연속 선을 구성하는 좌표를 기억할 자료형
struct PolyLineData
{
    int x[3], y[3]; // 삼각형을 구성하는 세 개의 좌표
};

// 선을 구성하는 좌표를 기억할 자료형
struct LineData
{
    int x[2], y[2]; // 선을 구성하는 두 개의 좌표
};
```

위 코드에서 사용한 구조체들은 **특정 도형을 위해 맞춤 제작된 한 덩어리의 데이터이기 때문에 재사용이 불가능한 구조**입니다. 그리고 이런 구조에서는 각 구조체의 공통점을 파악하거나 관계를 맺는 것도 힘들어 보입니다. 즉, 우리가 선언한 구조체는 도형을 구성하는 좌표 그 자체만 생각하고 만든 것이지, 도형이 가지는 좌표 데이터의 특징에 대해서는 전혀 고려를 하지 않았기 때문에 이런 문제가 생긴 것입니다.

문제를 해결하기 위해 구조체를 선언하면서 어떤 실수가 있었는지 다시 살펴보겠습니다. 먼저 우리가 선언한 구조체들은 좌표를 x, y라고 나누어 생각했는데, **한 점은 저장하는 좌표는 x, y가 하나의 그룹**이라서 아래와 같이 구조체로 정의해서 사용하

는 것이 맞습니다. 즉, 우리가 사용할 데이터들이 좌표를 기본으로 하고 있는데, 좌표에 해당하는 자료형도 선언하지 않고 시작했기 때문에 전체적으로 구조체들간의 공통점을 파악하기 힘들어진 것입니다.

```
// 한 점의 위치를 기억할 자료형
struct PointData
{
    int x, y;
};
```

이제 PointData 구조체를 사용해서 우리가 선언한 구조체를 수정해 보겠습니다.

```
#define MAX_COUNT   9

// 여러 개의 선으로 구성된 도형의 좌표를 기억할 자료형
struct MultiLineData
{
    int count;  // 선의 개수
    PointData start[MAX_COUNT]; // 선의 시작점
    PointData end[MAX_COUNT]; // 선의 끝점
};

// 연속 선을 구성하는 좌표를 기억할 자료형
struct PolyLineData
{
    PointData pos[3]; // 삼각형을 구성하는 세 개의 좌표
};

// 선을 구성하는 좌표를 기억할 자료형
struct LineData
{
    PointData start, end; // 선을 구성하는 두 개의 좌표
};
```

위와 같이 구성하면 LineData와 MultiLineData 구조체의 공통점이 보일 것입니다. 아래와 같이 LineData 구조체 선언을 위쪽으로 옮기고 MultiLineData 구조체는 LineData 구조체를 사용해서 재구성하면 됩니다. 이렇게 독자적인 구조체로 표현했던 자료형들의 공통점을 찾아 다든 자료형에 중첩된 표현을 사용하면 구조체 형식이 단순해져 자료형 간에 공통점을 찾기가 쉬워집니다.

```
// 한 점의 위치를 기억할 자료형
struct PointData
{
    int x, y;
};

// 선을 구성하는 좌표를 기억할 자료형
struct LineData
{
    PointData start, end; // 선을 구성하는 두 개의 좌표
};

#define MAX_COUNT   9

// 여러 개의 선으로 구성된 도형의 좌표를 기억할 자료형
struct MultiLineData
{
    int count;  // 선의 개수
    LineData line[MAX_COUNT]; // 도형을 구성하는 선 정보
};

// 연속 선을 구성하는 좌표를 기억할 자료형
struct PolyLineData
{
    PointData pos[3]; // 삼각형을 구성하는 세 개의 좌표
};

int main()
{
    MultiLineData data;

    // 첫 번째 선의 시작 좌표를 (5, 12)로 끝 좌표를 (120, 200)으로 설정
    data.count = 1;
    data.lines[0].start.x = 5;
    data.lines[0].start.y = 12;
    data.lines[0].end.x = 120;
    data.lines[0].end.y = 200;

    return 0;
}
```

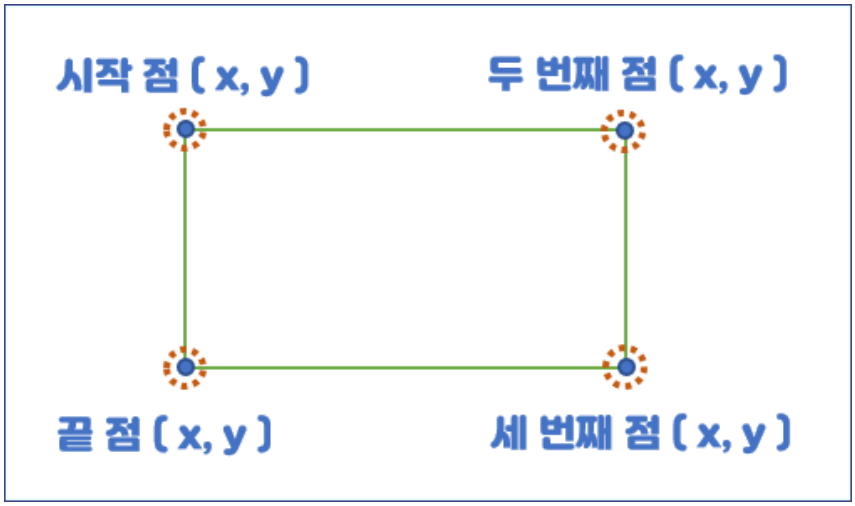
위 코드를 이해하기 편하도록 다시 정리해보면 다음과 같습니다.

◆ 한 개의 점은 두 개의 정숫값인 x, y 값으로 구성되기 때문에 x, y를 그둡으로 묶는 PointData 구조체를 선언
◆ 선은 두 개의 점을 연결한 것이므로 한 점을 저장하는 PointData를 두 개 사용해서 LineData 구조체를 선언
(LineData는 PointData를 사용하여 구조를 중첩해서 표현)
◆ MultiLineData 구조체는 선을 그둡으로 묶은 것이기 때문에 선 정보를 저장하는 LineData를 사용해서 선언
(MultiLineData는 LineData를 사용하여 구조를 중첩해서 표현)

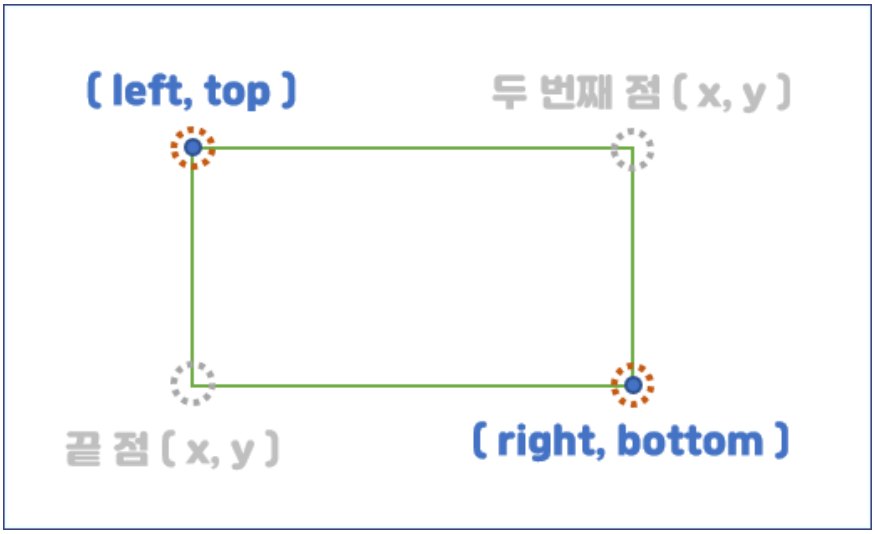
결론적으로 자료형을 표현할 때 개발자는 해당 자료형을 구성하는 세부 자료형들이 어떤 관계를 가졌는지 분석해서 데이터를 중첩(nested)해서 표현하는 것이 좋습니다. 이렇게 중첩해서 자료형을 표현해야지 데이터 형식에 변화가 생기더라도 좀 더 유연하게 대처할 수 있고 관련있는 자료형을 묶어서 공통 코드로 관리하는 것도 가능해집니다. 사실 어떤 점이 더 좋아지는지 자세하게 설명하고 싶지만 이 내용은 다형성(polymorphism) 문법과 연결되기 때문에 나머지는 다형성 강좌에서 설명하겠습니다.

2. 중첩된 표현은 사용하기 어려운 상황

중첩 표현이 좋다고 무조건 사용하라는 뜻은 아닙니다. 중첩이 가능한 구조라도 의미가 다르다면 중첩 표현을 무리하게 사용할 필요가 없습니다. 예를 들어, 사각형의 경우에는 네 개의 좌표로 위치를 표시할 것처럼 보이지만



아래와 같이 두 개의 좌표만 사용합니다. 왜냐하면 사각형의 특성상 두 번째 점으로 사용하는 좌표와 끝 점으로 사용하는 좌표는 나머지 두 좌표인 (left, top), (right, bottom)로 유추가 가능하기 때문입니다. 즉, 두 번째 점 (x, y)는 (right, top)이고 끝 점 (x, y)는 (left, bottom)입니다.



그래서 개발자들은 사각형 좌표를 구조체로 선언할 때, 위에서 설명한 의미를 유지하기 사각형의 좌표른 x, y 같은 이름이 아닌 left, top, right, bottom이라는 이름은 사용합니다.

```
// 사각형의 좌표를 저장할 자료형
struct Rect
{
    int left, top;
    int right, bottom;
};

int main()
{
    Rect pos;
    // (10, 15)과 (50, 60) 점을 연결한 사각형 좌표를 저장한다.
    pos.left = 10;
    pos.top = 15;
    pos.right = 50;
    pos.bottom = 60;
    return 0;
}
```

만약, 위와 같은 개념을 무시하고 사각형이 두 개의 좌표를 사용한다고 가정하고, 각각의 좌표는 Point 구조체로 중첩해서 아래와 같이 적으면 어떻게 될까요?

```
// 한 점의 위치를 기억할 자료형
struct Point
{
    int x, y;
};

// 사각형의 좌표를 저장할 자료형
struct Rect
{
    Point start, end;
};
```

일단, 위 코드는 문법적으로 전혀 문제가 되지 않습니다. 그리고 start로 사용한 (x, y) 좌표가 (left, top)을 의미하고 end로 사용한 (x, y) 좌표가 (right, bottom)을 의미한다는 것을 정확히 설명하고 사용하게 한다면 문제가 없을 것입니다. 하지만 이 상황을 잘 모르는 개발자가 봤을 때는 사각형의 좌표는 분명 네 개인데 이 중에서 start가 어떤 점을 의미하고 end가 어떤 점을 의미하는 것인지 오해하는 상황이 분명 생길 수 있습니다. 결국 **설명이 필요한 코드는 문제가 생기기 마련**입니다.

그렇다고 오해를 없애고자 아래와 같이 코드를 구성하면 어떻게 될까요?

```
// 한 점의 위치를 기억할 자료형
struct Point
{
    int x, y;
};

// 사각형의 좌표를 저장할 자료형
struct Rect
{
    Point left_top, right_bottom;
};
```

뭔가 Rect 구조체를 구성하는 두 좌표의 의미가 분명해지기는 했습니다. 하지만 이 구조체를 사용하는 코드를 아래와 같이 적어보면 문제가 있다는 것을 바로 이해할 수 있을 것입니다. pos.left_top.x = 10; 코드에 사용된 단어를 보면 좌표의 위치를 의미하는 단어가 left, top, x 이렇게 세 개나 사용되기 때문에 잘못된 좌표를 사용하는 실수를 할 수 있습니다. 이런 코드를 사용하면 꼼꼼한 개발자도 분명 버그가 생길 것입니다. 그러니 이렇게 표현에 문제가 될 수 있는 중첩된 구조체 표현은 사용하지 마세요.

```
int main()
{
    Rect pos;
    // (10, 15)과 (50, 60) 점을 연결한 사각형 좌표를 저장한다.
    pos.left_top.x = 10;
    pos.left_top.y = 15;
    pos.right_bottom.x = 50;
    pos.right_bottom.y = 60;
    return 0;
}
```

3. 중첩(nested)과 상속(inheritance)

중첩된 구조체 표현을 사용하면 데이터 변화에 대처하거나 소스 관리에 있어 장점이 많습니다. 하지만 중첩된 구조체를 사용하는 코드에 약간의 불편함이 생깁니다.

예를 들어, 아래와 같이 원 좌표를 기억하는 두 가지 구조체(Circle, DirectCircle)를 선언했는데 Circle 구조체는 중첩 표현을 사용했고 DirectCircle 구조체는 직접 표현을 사용했습니다. 그리고 두 구조체를 사용해서 pos1, pos2 변수를 선언하고 중심점과 반지름 정보를 저장하는 코드를 작성했습니다. main 함수에 작성된 코드를 보면 pos1.center.x에 사용된 항목 지정 연산자(.)가 pos2.circle_x 에 사용된 '_'로 바뀌었을 정도로 코드가 매우 유사합니다.

```
// 한 점의 위치를 기억할 자료형
struct Point
{
    int x, y;
};

// 원을 구성하는 좌표를 기억할 자료형
struct Circle
{
    Point center; // 중심점
    int radius;   // 반지름
};

// 원을 구성하는 좌표를 기억할 자료형
struct DirectCircle
{
    int center_x, center_y; // 중심점
    int radius;             // 반지름
};

int main()
{
    Circle pos1;
    DirectCircle pos2;

    // 중심점이 (50, 60)이고 반지름이 15인 원
    pos1.center.x = 50;
    pos1.center.y = 60;
    pos1.radius = 15;

    // 중심점이 (150, 160)이고 반지름이 20인 원
    pos2.center_x = 150;
    pos2.center_y = 160;
    pos2.radius = 20;

    return 0;
}
```

그런데 위에서 선언한 구조체들이 원을 표현하는 자료형이기 때문에 좌표를 의미하는 하나의 변수가 선언되면 해당 좌표가 중심점이라고 생각하기 마련입니다. 그래서 코드를 간략하게 표현하기 위해 center라는 단어를 생략하려면 아래와 같이 DirectCircle 구조체의 경우에는 center_x를 x로 center_y를 y로 줄여서 사용하는 것이 가능합니다. 하지만 Circle 구조체는 중첩된 표현을 사용하기 때문에 center를 생략할 수 없습니다. 그래서 간략한 표현을 사용하고 싶어도 최소한의 의미를 남기려면 pos로 줄이는 것이 최선입니다.

결국 중첩된 표현은 다시 세부 항목을 선택하기 위해 항목 지정 연산자(.)를 사용해야 하기 때문에 이름이 길게 나열되는 단점이 있습니다. 그리고 지금은 중첩이 한 번만 되었기 때문에 항목 지정 연산자가 한 번만 사용되지만, 중첩이 세 번되었다면 구조체를 사용할 때 원하는 항목을 호출하기 위해서는 항목 지정 연산자를 세 번이나 사용해야 한다는 뜻입니다.

```cpp
// 한 점의 위치를 기억할 자료형
struct Point
{
    int x, y;
};

// 원을 구성하는 좌표를 기억할 자료형
struct Circle
{
    Point pos; // 중심점
    int radius;   // 반지름
};

// 원을 구성하는 좌표를 기억할 자료형
struct DirectCircle
{
    int x, y;       // 중심점
    int radius;   // 반지름
};

int main()
{
    Circle pos1;
    DirectCircle pos2;

    // 중심점이 (50, 60)이고 반지름이 15인 원
    pos1.pos.x = 50;
    pos1.pos.y = 60;
    pos1.radius = 15;

    // 중심점이 (150, 160)이고 반지름이 20인 원
    pos2.x = 150;
    pos2.y = 160;
    pos2.radius = 20;

    return 0;
}
```

그렇다면 대부분의 잘 계획된 프로그램에서 구조체들은 중첩된 표현을 많이 사용할텐데, 이렇게 선언된 구조체를 사용할 때마다 **항목 지정 연산자를 너무 많이 사용해서 분편했었다면 C++ 언어에서 제공하는 상속(inheritance) 문법은 사용**하면 됩니다.

```
// 한 점의 위치를 기억할 자료형
struct Point
{
    int x, y;
};

// 원을 구성하는 좌표를 기억할 자료형
struct Circle
{
    Point pos; // 중심점
    int radius;    // 반지름
};
```

예를 들어, 위와 같이 사용되는 것이 중첩 표현이라면, 아래와 같이 Circle 구조체를 선언할 때 구조체의 이름 뒤에 콜론(:)은 적고 그 뒤에 복사할 구조체의 이름인 Point를 적으면 Circle 구조체가 Point 구조체로부터 상속된 것은 의미합니다. 이렇게 상속 구조가 만들어지면 Point 구조체른 부모 구조체라 이야기하고 Circle 구조체는 자식 구조체라 이야기합니다.

상속이라는 표현을 너무 어렵게 생각하는 개발자들이 많은데, 상속은 단순히 구조체(클래스)의 내용은 복사하는 기술입니다. 즉, Circle 구조체에는 int x, y;라고 선언되지 않았지만 상속은 통해 Point에서 int x, y;가 복사되었기 때문에 Circle 구조체는 DirectCircle 구조체와 동일한 구조가 된 것입니다. 그래서 main 함수에 사용된 코드를 보면 pos1과 pos2 변수는 동일한 형태로 항목에 값을 대입하는 것을 볼 수 있습니다.

```
// 한 점의 위치를 기억할 자료형 (부모 구조체)
struct Point
{
    int x, y;
};

// 원을 구성하는 좌표를 기억할 자료형, Point 구조체에서 상속 됨 (자식 구조체)
struct Circle : Point
{
    int radius;    // 반지름
};

// 원을 구성하는 좌표를 기억할 자료형
struct DirectCircle
{
    int x, y; // 중심점
    int radius;              // 반지름
};

int main()
{
    Circle pos1;
    DirectCircle pos2;

    // 중심점이 (50, 60)이고 반지름이 15인 원
    pos1.x = 50;
    pos1.y = 60;
    pos1.radius = 15;

    // 중심점이 (150, 160)이고 반지름이 20인 원
    pos2.x = 150;
    pos2.y = 160;
    pos2.radius = 20;

    return 0;
}
```

결국 **상속은 사용하면 구조를 복사하는 개념이 적용되어 상속된 구조체에 부모 구조체의 모든 항목이 복사**됩니다. 따라서 구조체에 해당 항목들을 직접 선언한 것과 동일한 구조가 되어 해당 항목을 사용할 때 **항목 지정 연산자를 추가로 사용할 필요가 없습니다.** 이것은 코드만 간결해지는 것이 아니라 실제로 수행 능력에도 영향을 미치기 때문에 C++ 언어에서는 중첩 표현보다는 상속 표현을 더 많이 사용합니다.

그리고 상속을 설명하는 분들 중에 **'자식 구조체가 부모 구조체를 호출한다'는 식의 표현을 사용하는분들이 있는데 이것은 잘못된 표현**입니다. 다시 한 번 이야기하지만 상속은 구조체(클래스)를 복사하는 기술이고 상속을 사용하면 지정된 부모 구조체(클래스)의 내용이 그대로 자식 구조체(클래스)에 포함됩니다.

댓글 등록순 최신순

관심글 댓글 알림

해적왕곰팅 3

꼭 알아야 되는 기본적인 자료형 설계에 대해 깨우치고 가네요~ㅎ 감사합니다.

2022.06.02. 02:10 답글쓰기

김성엽 M 작성자

늦은밤에 열공하시는군요 ㅎ

2022.06.02. 02:15 답글쓰기

조민희 3

상속은 부모 구조체의 내용을 자식 구조체에 복사하는 기술이다' 좋은 강의 감사드립니다 ㅎㅎ

2022.07.19. 22:43 답글쓰기

김성엽 M 작성자

꾸준하게 보시는군요 ㅎㅎ 파이팅입니다!

2022.07.19. 23:42 답글쓰기

카일 3

강의 잘 들었습니다. 말씀대로 볼 때마다 생각이 확장됩니다.

2022.11.25. 21:23 답글쓰기

김성엽 M 작성자

2022.11.25. 21:52 답글쓰기

dh221009

댓글을 남겨보세요

등록