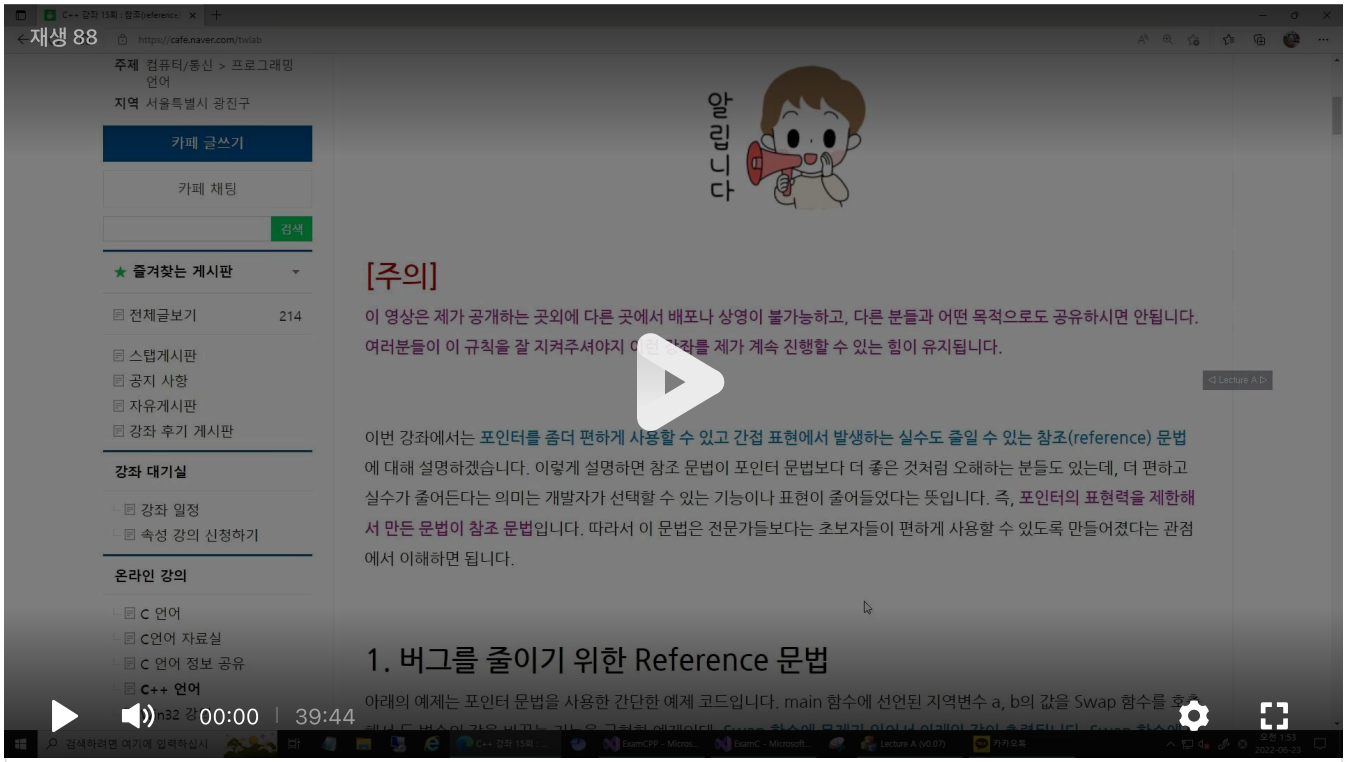
C++ 강좌 15회 : 참조(reference) 문법

김성엽 카페매니저 + 구독 1:1 채팅
2022.06.18. 18:46 조회 289

[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 **포인터를 좀더 편하게 사용할 수 있고 간접 표현에서 발생하는 실수도 줄일 수 있는 참조(reference) 문법**에 대해 설명하겠습니다. 이렇게 설명하면 참조 문법이 포인터 문법보다 더 좋은 것처럼 오해하는 분들도 있는데, 더 편하고 실수가 줄어든다는 의미는 개발자가 선택할 수 있는 기능이나 표현이 줄어들었다는 뜻입니다. 즉, **포인터의 표현력은 제한해서 만든 문법이 참조 문법**입니다. 따라서 이 문법은 전문가들보다는 초보자들이 편하게 사용할 수 있도록 만들어졌다는 관점에서 이해하면 됩니다.



C++ 15회차 강좌 - 참조 문법

1. 버그를 줍이기 위한 Reference 문법

아래의 예제는 포인터 문법을 사용한 간단한 예제 코드입니다. main 함수에 선언된 지역변수 a, b의 값을 Swap 함수를 호출해서 두 변수의 값을 바꾸는 기능을 구현한 예제인데, Swap 함수에 문제가 있어서 아래와 같이 춘력됩니다. Swap 함수에 어떤 문제가 있는지 찾아보세요. 당연한 이야기겠지만 아래의 코드는 문법적으로는 문제가 없는 코드라서 컴파일도 잘 되고 실행도 잘됩니다.

```
#include <stdio.h>

void Swap(int *pa, int *pb)
{
    int temp = *pa;
    pa = pb;
    *pb = temp;
}

int main( )
{
    int a = 5, b = 6;
    printf("before : a = %d, b = %d\n", a, b);
    Swap(&a, &b);
    printf("after : a = %d, b = %d\n", a, b);

    return 0;
}
```

위 예제는 아래와 같이 출력됩니다. 정상적으로 구현되었다면 a는 6, b는 5가 춘력되어야 합니다.

```
before : a = 5, b = 6
after : a = 5, b = 5
```

잘못된 부분을 찾으셨나요? 정답은 아래와 같습니다. *pa = *pb; 처럼 코드를 구성해야 하는데 * 연산자를 실수로 생략해서 pa = pb;라고 사용해서 생긴 문제입니다. 즉, pb 포인터가 가리키는 대상(main 함수의 b 변수)의 값을 pa 포인터가 가리키는 대상(main 함수의 a 변수)에 저장하는 코드를 사용해야 하는데 pb 변수에 저장된 주소를 pa 변수에 그냥 대입해서 발생한 문제입니다.

```
void Swap(int *pa, int *pb)
{
    int temp = *pa;
    *pa = *pb;      // 이 부분에 문제가 있었습니다. (pa = pb;라고 되어 있었음)
    *pb = temp;
}
```

이런 실수가 발생하는 이유는 아래와 같이 포인터 변수를 사용하는 방법이 두 가지 경우로 나누어져 있기 때문입니다. 이렇게 포인터가 두 가지 표현을 사용하는게 문제라는 뜻이 아니라 어떤 문법이든 상황은 선택하는 기능이 포함되면 초보자들이 어려워하고 실수도 하게 된다는 의미입니다.

```
1. 포인터 변수에 저장된 주소를 읽거나 쓰는 방법 :  * 연산자를 사용하지 않는다.
2. 포인터 변수가 가리키는 대상의 값을 읽거나 쓰는 방법 :  * 연산자를 함께 사용한다.
```

결국 개발자가 2번 형식으로 사용해야 하는데 1번 형식으로 사용해서 이런 문제가 발생한 것이고 초보자이기 때문에 이런 실수를 할 수도 있다는 의미입니다. 하지만 초보자들이 어려워하고 실수한다는 이유로 포인터 두 가지 표현 중에 한 가지 표현을 제거하는 것은 좋은 선택이 아닙니다. 그래서 C++ 언어는 포인터 문법은 그대로 두고 초보자들이 좀더 코드를 쉽게 이해하고 실수도 하지 않도록 위 두 가지 상황에서 1번 표현을 제거한 참조(reference) 문법을 추가로 제공합니다.

예를 들어, 정수 변수의 값을 포인터 문법을 사용해서 변경하는 코드는 다음과 같습니다.

```
int data = 5;
int *p = &data;  // data 변수의 주소를 포인터 변수 p에 저장
*p = 7;  // p가 가리키는 대상에 7을 대입. data 변수의 값이 5에서 7로 변경
```

그리고 위 코드를 참조 문법을 사용해서 적어보면 다음과 같습니다. 위 코드와 비교해보면 & 연산자는 동일하게 사용되었지만 * 연산자가 사라진 것을 볼 수 있습니다. 물론 위에서 사용된 &는 번지 계산 연산자이고 아래에 사용된 &는 참조 변수를 의미하는 자료형(int &) 표현이기 때문에 동일한 표현은 아닙니다. 그리고 위 표현에서는 포인터 변수 p를 사용할 때 p = 7; 이라고 사용하는 것도 가능하고 *p = 7; 이라고 사용하는 것도 가능하기 때문에 개발자가 어떤 기능을 사용할 것인지 선택해야 합니다. 하지만 아래의 참조 표현에서는 r 변수가 참조하는 대상의 값만 수정이 가능하기 때문에 r = 7; 표현 밖에 사용하지 못합니다. 따라서 개발자가 참조 문법을 사용할 때는 상황 선택을 위한 고민을 하지 않아도 됩니다. 즉, 위 표현에서는 1, 2번 상황 중에 어떤 상황을 사용해야 할 지 개발자가 고민해야 하지만 아래의 표현에서는 2번 상황만 사용이 가능하기 때문에 고민할 필요가 없다는 뜻입니다.

```
int data = 5;
int &r = data;  // r 변수는 data 변수를 참조하도록 지정
r = 7;  // r 변수가 참조하는 대상에 7을 대입. data 변수의 값이 5에서 7로 변경
```

결국 참조 변수를 선언하면 대상의 값만 사용하겠다는 의미입니다. 그래서 참조 변수 r을 사용할 때는 포인터처럼 번지 지정(*) 연산자를 사용하지 않고 변수 이름만 사용하게 됩니다. 그리고 이렇게 연산자 사용 여부를 선택하지 않는 표현은 오직 한 가지 형태의 표현만 사용하기 때문에 버그가 발생할 확률도 줄여줍니다.

2. 포인터를 사용할 때 발생하는 버그를 줄이는 두 가지 방법

1번에서 소개한 Swap 함수에서는 포인터 변수가 가리키는 대상의 값만 사용하기 때문에 포인터 변수에 저장된 주소를 변경할 이유가 없습니다. 그리고 만약 포인터 변수의 주소가 변경된다면 버그 상황입니다. 그래서 C 언어는 Swap 함수내에서 실수로 포인터 변수의 주소가 변경되는 것을 막기 위해 아래와 같이 const 키워드 사용을 권장합니다. 즉, 아래와 같이 포인터

변수 pa, pb를 const 키워드로 지정하면 pa, pb 값은 읽기만 가능합니다. 따라서 실수로 pa = pb;라고 코드를 적으면 pa 변수값을 변경할 수 없는데 변경하려고 하려고 시도했기 때문에 컴파일할 때 문법 오류가 발생합니다.

```
void Swap(int * const pa, int * const pb)
{
    int temp = *pa;
    *pa = *pb;       // 만약, 실수로 pa = pb; 라고 적으면 pa가 const로 지정되어 컴파일 오류 발생!
    *pb = temp;
}
```

하지만 위와 같은 표현이 버그를 막는 정석적인 대처 방법이지만 포인터를 어려워하는 초보 개발자들에게 const 키워드까지 사용하게 하는 것도 좋은 방법은 아니라고 생각합니다. 그래서 C++ 언어는 초보자들을 위해 위와 같은 상황에서 사용할 수 있는 참조 문법을 추가로 제공하는 것이고 Swap 함수에 참조 문법을 적용하면 아래와 같이 됩니다. 결국 **어떤 기능이 한 가지 목적으로 사용되었다면 선택적인 표현은 사용하는 문법보다는 전용 문법은 제공하는 것이 초보 개발자들이 더 쉽게 느끼고 실수할 확률도 줄여준다**는 뜻입니다.

```
#include <stdio.h>

void Swap(int &ra, int &rb)
{
    int temp = ra;
    ra = rb;         // 실수가 발생할 확률이 없음!
    rb = temp;
}

int main( )
{
    int a = 5, b = 6;
    printf("before : a = %d, b = %d\n", a, b);
    Swap(a, b);  // 참조 문법으로 주소를 전달하기 때문에 호출할 때 &를 사용하지 않는다.
    printf("after : a = %d, b = %d\n", a, b);

    return 0;
}
```

위 예제는 아래와 같이 출력됩니다.

```
before : a = 5, b = 6
after : a = 6, b = 5
```

3. 참조 변수도 대상의 주소를 가지고 있다.

아래와 같이 참조 변수를 사용했다면 r 변수에는 data 변수를 참조하기 위해 data 변수의 주소가 저장되어 있습니다. 따라서 **r = 5; 라고 사용한 코드는 개발자가 선택만 못할뿐, 내부적으로는 번지 지정 연산자가 무조건 적용되어 data 변수에 5가 대입**되는 것입니다.

```
int data = 1;
int &r = data;  // r 변수는 data 변수를 참조해서 사용
r = 5;  // data 변수에 5가 대입 됨
```

따라서 data 변수를 포인터 변수 p에 저장할 때는 &data를 사용해서 주소를 얻을 수도 있지만 아래와 같이 r 변수에 저장된 주솟값을 받아도 됩니다. 이때 **&r은 r 변수의 주소를 계산하는 것이 아니라 r 변수가 참조하는 변수의 주소를 계산한다는 의미라서 data 변수의 주소를 얻게 됩니다.** 실제로 p = &r; 코드는 기계어로 번역되면 r 변수에 저장된 값을 그대로 p에 대입하는 코드로 번역됩니다.

```
int data = 1;
int &r = data;  // r 변수는 data 변수를 참조해서 사용
r = 5;  // data 변수에 5가 대입 됨

int *p;
p = &r;  // r 변수에 저장된 주솟값을 p에 대입한다. (data 변수의 주소가 p에 대입)
*p = 10;  // data 변수에 10이 대입 됨
```

위와 같은 코드에서는 &r 대신 &data를 사용해서도 data 변수의 주소를 얻을 수 있기 때문에 상관없지만 아래와 같이 참조 변수가 매개 변수에 사용된 경우에는 r 변수가 참조하는 data 변수의 주소를 r 변수를 사용해서만 알 수 있기 때문에 &r 같은 표현을 알고 있는 것이 좋습니다. 즉, **실제 변수의 주소가 참조 변수에 감추어진 상태로 함수에 전달되더라도 해당 변수의 주소를 사용하고 싶다면 &r이라고 사용하면 된다는 뜻**입니다.

```
void Test(int &r)
{
    int *p;
    p = &r;  // r 변수가 참조하는 변수의 주소가 p에 대입
    *p = 10;  // main 함수의 data 변수에 10이 대입
}

int main()
{
    int data = 1;
    Test(data);

    return 0;
}
```

4. 참조 변수로 배열은 참조할 수 있는가?

참조 변수도 참조 대상의 자료형 기준으로 대상을 사용할 수 있습니다. 예를 들어, int &r;이라고 참조 변수를 선언하면 참조 변수 r은 int 자료형으로 선언된 대상을 참조하게 됩니다. 그래서 int &r;이라고 참조 변수를 선언하면 정숫값 한 개에 해당하는 메모리만 참조가 가능하기 때문에 참조 대상을 temp[0] 처럼 배열의 한 개 항목만 사용할 수 있습니다.

```
int temp[5] = { 1, 2, 3, 4, 5 };
int &r = temp[0];
r = 10;  // temp 배열의 0번 항목 값이 10으로 변경 됨
```

만약, 참조 변수로 배열 전체를 다 사용하고 싶다면 아래와 같이 참조 변수를 선언할 때 참조 대상의 자료형에 참조할 배열의 자료형을 그대로 적어주면 됩니다. 결국 형식만 조금 다를뿐 기본 개념은 포인터와 동일합니다.

```
int temp[5] = { 1, 2, 3, 4, 5 };
int (&r)[5] = temp;
r[3] = 10;  // temp 배열의 3번 항목 값이 10으로 변경 됨. ( temp[3] = 10; )
```

5. 참조 변수를 사용할 때 주의해야 할 점

참조 변수를 사용하는 개발자들이 초보자인 경우가 많다보니, 아래와 같이 참조 변수 형식으로 값을 반환해서 사용하는 경우가 종종 있습니다. 물론 이 코드는 정상적으로 동작하기 때문에 이 코드에 문제가 있다는 것을 모르고 사용하는 분들이 많습니다. 앞에서 계속 이야기 했지만 참조 문법은 포인터 기술을 사용하고 있습니다. 따라서 TestFunc 함수가 return num;으로 num 변수의 값을 반환하는 것처럼 구성되었지만, 함수의 반환 자료형이 int &이기 때문에 실제로는 참조 문법이 적용되어 num 변수의 주소가 반환되는 형식입니다. 그런데 num 변수가 TestFunc 함수의 지역 변수이기 때문에 TestFunc 함수가 작업을 완료하면 함께 사라지는 변수입니다. 그런데 개발자는 사라지게 된 num 변수의 주소를 반환해서 사용하기 때문에 이 코드는 안정성을 보장받을 수 없는 버그 코드인 것입니다.

```
#include <stdio.h>    // printf 함수를 사용하기 위해

int &TestFunc()
{
    int num = 5;
    return num;
}

int main()
{
    int &r = TestFunc();
    printf("%d\n", r);

    return 0;
}
```

위 예제는 아래와 같이 정상적으로 5가 출력됩니다.

```
5

이 창을 닫으려면 아무 키나 누르세요...
```

그런데 위 코드는 어떻게 정상적으로 잘 출력된 것일까요? 그 이유는 매우 단순합니다. r 변수를 사용해서 값을 출력하는 시점에서 본때 TestFunc 함수가 사용했던 메모리 공간을 다른 함수가 아직 점유하지 않았기 때문입니다. 즉, 운좋게 다른 함수가 호출되지 않은 상태에서 r 변수를 사용했기 때문에 이전에 사용했던 num 변수 값이 다른 함수에 의해 덮어쓰여지지 않아 정상적으로 출력된 것입니다.

하지만 아래와 같이 r 변수의 값을 출력하는 코드를 두 번 연속으로 적으면 문제가 발생합니다. printf 함수를 사용한 첫 출력에서는 정상적으로 5가 출력되지만, 두 번째 출력할 때는 이상한 값이 출력됩니다.

이것은 첫 번째 r 변숫값을 출력할 때는 아직 다른 함수가 호출된 적이 없어 TestFunc 함수가 사용했던 메모리 공간이 이전 상태로 그대로 유지되었기 때문에 r 변수가 참조하고 있는 값이 정상적으로 출력된 것입니다. 그리고 이 작업이 진행되면서 기존에 TestFunc 함수가 사용하던 영역에 printf 함수의 지역 변수 값들로 덮어쓰기가 되어 버립니다. 그래서 두 번째 r 변숫값을 출력하는 코드에서는 num 변수가 있던 자리에 덮어쓰기된 잘못된 값을 출력하기 때문에 원하지 않는 결과가 출력된 것입니다.

```
#include <stdio.h>    // printf 함수를 사용하기 위해

int &TestFunc()
{
    int num = 5;
    return num;
}

int main()
{
    int &r = TestFunc();
    printf("%d\n", r);  // TestFunc 함수가 사용하던 영역이 보존됨
    printf("%d\n", r);  // 위 printf 함수가 호출되어 TestFunc 함수 영역이 덮어쓰기가 되어버림

    return 0;
}
```

위 예제는 아래와 같이 출력됩니다.

```
5
1266287

이 창을 닫으려면 아무 키나 누르세요...
```

참조 문법을 사용하는 사람들중에 초보 개발자가 많다보니 의외로 위와 같은 잠재적 버그 코드를 사용하는 사람들이 많습니다. 그리고 이런 버그는 간단한 코드에서는 나타나지 않고 프로그램이 복잡해지면 나타나기 때문에 문제가 발생하게 되면 해

결하기가 더 어렵습니다. 따라서 가능하면 참조 형식으로 반환을 사용하지 않는 것이 좋습니다.

6. 포인터 문법과 참조 문법

두 문법은 사실 비교 대상이 아닙니다. 참조 문법은 포인터 문법의 간소화된 표현일 뿐이기 때문에 포인터 문법을 잘 사용하는 개발자는 사실상 참조 문법을 사용하지 않습니다. 그리고 위에서 소개했던 Swap 함수의 버그 상황도 사실상 포인터에 익숙하지 않은 초보자에게 발생하는 문제이기 때문에 초보자 수준을 벗어나면 사실상 이런 버그는 나오지 않는다고 보면 됩니다.

하지만 모든 개발자가 전문가는 아니기 때문에 함께 작업하는 개발자가 초보자라면 실수를 방지하는 차원에서 또는 초보 개발자를 배려하는 차원에서 참조 문법을 사용하는 것도 괜찮다고 생각합니다. 그리고 **참조 문법과 포인터 문법은 실제 기계어로 번역되면 동일하기 때문에 성능 저하도 발생하지 않습니다.** (최적화 옵션 선택에 따라 차이가 있습니다.)

마지막으로 개인적인 의견을 추가하면 C, C++ 문법이 어려운 이유는 아직 개발자가 그 문법에 익숙하지 않아서 그런 것입니다. 그런데 그 어려움을 회피하기 위해 초보자들 문법을 사용하게 되면 그냥 초보자스러운 코드만 사용하게 되어 전문가로 가는 길이 더 멀어질 것입니다. 어차피 전문 개발자가 되려면 그 어려움을 정면으로 돌파해야 하기 때문에 조금 어렵게 느껴지더라도 해당 문법들을 반복해서 사용하는 것이 좋습니다. 이렇게 반복해서 사용하다보면 문법들이 익숙해질 것이고 그러면 얼마가지 않아 문법이 쉬워지는 환상적인 경험을 하게 될 것입니다.

으악 포인터만 알고 있다가 참조 개념을 보니 포인터가 다시 헷갈리네요 ㅠㅠ
남들이 참조 표현을 썼을때 읽을 수 있을 정도로만 공부하고, 쓰던데로 포인터 쓰겠습니다!!

2022.09.12. 16:39 답글쓰기

김성엽 작성자

네. 그게 좋습니다. 섞어쓰면 더 헷갈려요 ㅎ

2022.09.12. 16:54 답글쓰기

dh221009

댓글을 남겨보세요

등록