Safe C++

C++은 포인터와 null(nullptr)이 사용되는 언어이다. 이는 C#이나 Go와 같은 언어도 마찬가지이다. C#이나 Go와 다른 점은 C++은 직접 메모리를 관리하기 때문에 오류의 가능성이 좀 더 높다. C#이나 Go는 Garbage Collection(GC)을 사용하기 때문에 참조를 갖고 있으면 해제되지 않는다. 메모리 증가는 있어도 죽지는 않는다. 물론 NullReferenceException이 생길 수 있다. null이 있는 언어는 모두 이와 같은 위험성을 내포하고 있지마 C++만큼 심하지는 않다.

참조 카운트 기반의 메모리 관리

std 포인터들

GC 만큼 안전한 참조 카운트 기반의 메모리 관리를 한다. std::shared_ptr와 std::unique_ptr, 그리고 필요할 경우 std::weak_ptr을 사용하면 포인터 사용 자체는 GC가 있는 언어만큼 안전해진다.

참조 사용

이에 더해 컨벤션으로 참조 사용이 가능할 경우 참조를 사용하면 된다. 소유권을 명시적으로 관리한다. 누가 소유하는지를 분명하게 한다. 참조로 구성할 수 없는 경우 포인터를 직접 얻을 수 있는데 이런 경우 std::shared_ptr을 사용하거나 자체적으로 안전한 포인터를 구성하는 게 좋다. 초기화 지점을 두고 해당 지점을 통과했다면 항상 안전하다는 걸 보장할 방법이 있으면 더욱 좋다.

컨테이너에서 찾아서 사용하는 경우 반드시 체크가 항상 있어야 한다. 포인터를 찾지 않고 참조를 얻는 방식이 좋다. Has와 Find를 구분하여 처리하는 것이 좋다.

```
return_if(!Container.Has(id));

auto& elem = container.Get(id);
/// ... 처리 코드
```

mark and purge

포인터 소멸 시 참조가 먼저 제거되도록 구성해야 한다. Unreal 4에서는 주요 객체인 Actor들에 대해 Pending Kill 상태를 추가하고 다음 프레임에서 제거하도록 한다. Actor 내의 다양한 오브젝트들에서 실행 중 무효화되는 것을 방지하기 위한 방법이다. 컨테이너 순회 중 제거시에도 이와 같은 기법이 필요하기 때문에 제거된 상태로 마킹만 하고 다음 처리에서 모두 제거하는 (purge) 방법이 필요하다. mark and purge라고 이름 지을 수 있겠다.

포인터 사용 컨벤션

std::shared_ptr을 포함한 참조 기반 포인터들도 nullptr일 수 있다. 따라서, 포인터를 사용하기 전에는 항상 모두 체크해야 한다. 처리 블럭 전 단계에서 모두 체크해야 한다. assert를 요구할 수 있는 경우도 있고 아닌 경우도 있지만 모두 체크하는 것이 필요하다.

이를 위해 assert, return_if() 와 같은 매크로를 추가하여 점검과 처리 루틴을 포함해야 한다. RAII와 함께 사용하여 코드 실행 중 획득한 자원들이 자동으로 반환되도록 구성해야 한다.

조건 체크의 습관화

깊은 들여쓰기 (deep nested indentation)를 피하는 방법이 서버 코드에는 좀 더 적절해 보인다. Unreal 4의 경우 if 문의 포함을 사용하여 서술하는데 서버의 경우 에러 조건을 체크하고 일찍 나가는 것이 코드를 보기가 더 낫고 실수를 줄일 수 있다.

함수 진입 시 아규먼트와 클래스 상태에 대한 조건 쳄크를 둔다. assert 가 있으면 에러 체크도 함께 두도록 한다. 서버의 경우 assert의 조건이 깨지면 서버의 크래시가 발생하고 테스트 코드 작성이 코드 커버리지가 되는 수준으로 작성되지 않는 현실을 고려하여 예외적인 경우가 아니라면 에러 처리를 함께 한다. 특히, 포인터 사용에 대해서는 꼭 포함하도록 한다.

안전한 로그와 문자열 사용

C++의 대부분 런타임은 로그 때문에 크래시가 날 수 있다. 이런 문제를 해결하기 위해 안전한 로그나 문자열 처리 라이브러리들이 많이 나왔다. 현재는 fmt 라이브러리가 매우 빠르면서도 C++ 11 이후에 맞춰 사용성이 좋도록 나와 있다.

auto& str = fmt::format("{0} is {1}", "He", "a man"); 과 같이 사용할 수 있다. 아규먼트가 빠지면 빠진 사실만 통보해주고 nullptr이 넘어오면 nullptr이라고 알려준다. 죽지 않는다.

사용도 printf와 거의 동일하므로 특별히 문제가 없고 std::wstring은 utf8 전환을 통해 지원한다. 나중에 로그 등에 넣기는 쉽지 않을 수 있지만 시도해 볼 가치는 있다.

std::array의 사용

std::vector를 사용하면 인덱스 체크를 거의 항상 한다. C++ 배열의 경우 누락하기 쉽다. std::array가 더 안전한가?

- size() 함수로 항상 크기를 알 수 있다.
- iteartor로 순회할 수 있고 ranged for 문을 쓸 수 있다.

위 두 가지로 vector 만큼 안전하게 사용할 수 있다. 참조형으로 돌아다니면 성능 저하도 없다. 정적으로 크기가 할당되므로 메모리 오버헤드도 없다. 따라서, std::array를 사용할 수 있는 곳은 std::array를 쓰는 것이 좋겠다.

std::vector 대용으로 사용하는 건 어떨까? 삽입 / 삭제가 많을 경우 체크가 필요하여 std::vector가 더 낫겠다.

C++의 배열은 사용하지 않고 std::array를 사용한다. 클래스 내부 구현에만 딱 필요한 경우 C++ 배열을 사용할 수는 있겠다. 개인적으로 std::vector 외에 배열을 사용한 경우는 거의 없는 듯 하다. 크기가 딱 정해진 경우만 std::array를 사용했다.