**항목 26. 변수 정의는 늦출 수 있는 데까지 늦추는 근성을 발휘하자.**

생성자 혹은 소멸자를 끌고 다니는 타입으로 변수를 정의하면, 반드시 물게 되는 비용이 2개 있습니다.   
하나는 프로그램 제어 흐름이 **변수의 정의에 닿을 때 생성자가 호출되는 비용**이고,  
또 하나는 그 변수가 유효범위를 벗어날 때 **소멸자가 호출되는 비용**입니다.  
변수가 정의됐으나 사용되지 않은 경우에도 비용이 부과되는데, 이런 비용은 웬만한 경우가 아니면 물고 싶을 생각이 안 들 것입니다.

아래 예,  
이 함수는 주어진 비밀번호가 충분히 길 경우에 비밀번호를 암호화하여 반환하는 함수입니다.  
비밀번호가 너무 짧으면 logic\_error 타입의 예외를 던지도록 만들어졌는데,   
logic\_error 타입은 표준 C++ 라이브러리에 정의되어 있습니다 (54)

|  |
| --- |
| // 이 함수는 "encrypted" 변수를 너무 일찍 정의해 버립니다.  std::string encryptPassword(const std::string& password)  {             using namespace std;    **string encrypted;**               if (password.length() < MinimumPasswordLength) {                       throw logic\_error("password is too short");             }             ... // 주어진 비밀번호를 암호화하여 encrypted 변수에 넣는데 필요한 일들을 여기서 합니다.               return encrypted;  } |

Encrypted 객체가 이 함수에서 **완전히** 안 쓰인다고는 말할 수 없지만, 예외가 발생되면 이 변수는 분명히 사용되지 않게 됩니다. 즉, encryptPasword 함수가 예외를 던지더라도 encrypted 객체의 생성과 소멸에 대해 비용을 내야 한다는 이야기입니다.

이런 사정을 확인한 이상, encrypted 변수를 정의하는 일은 꼭 필요해지기 전까지로 미루는 편이 낫겠다는 생각이 들겠지요.

|  |
| --- |
| // 이 함수는 "encrypted" 변수가 진짜로 필요해질 때까지 정의를 미룹니다.  std::string encryptPassword(const std::string& password)  {             using namespace std;               if (password.length() < MinimumPasswordLength) {                       throw logic\_error("password is too short");             }    **string encrypted;**     ... // 주어진 비밀번호를 암호화하여 encrypted 변수에 넣는데 필요한 일들을 여기서 합니다.               return encrypted;  } |

Encrypted 변수가 정의될 때 초기화 인자가 하나도 없습니다. 기본 생성자가 호출될 거란 뜻이지요.  
객체를 기본 생성하고 나서, 값을 대입하는 방법이 어째서 여러분이 원하는 값으로 직접 초기화하는 방법보다 효율이 좋지 않은지는 (4)를 확인.

|  |
| --- |
| // "encrypted" 를 정의하고 초기화하는 가장 좋은 방법.  std::string encryptPassword(const std::string& password)  {             ... // 길이를 점검합니다.               std::string encrypted(password); // 변수를 정의함과 동시에 초기화합니다.               encrypt(encrypted);             return encrypted;  } |

어떤 변수를 사용해야 할 때가 오기 전까지, 그 변수의 정의를 늦추는 것은 기본이고,   
초기화 인자를 손에 넣기 전까지 정의를 늦출 수 있는지도 둘러봐야 한다.

이렇게 해야 쓰지도 않을 객체가 만들어졌다 없어지는 일이 생기지 않으며, 불필요한 기본 생성자 호출도 일어나지 않습니다. 덤으로, 누가 보아도 그 변수의 의미가 명확한 상황에서 초기화가 이루어지기 때문에, 변수의 쓰임새를 문서화하는데도 도움이 됩니다.

루프에 대해서는?  
어떤 변수가 루프 안에서만 쓰이는 경우라면,   
해당 변수를 루프 바깥에서 미리 정의해 놓고 루프 안에서 대입하는 방법이 좋을까요,   
아니면 루프 안에 변수를 정의하는 방법이 좋을까요?

|  |
| --- |
| // **A 방법**: 루프 바깥쪽에 정의  **Widget w;**  for (int i = 0; i < n; i++) {  **w =** i에 따라 달라지는 값;             ...  }    // **B 방법**: 루프 안쪽에 정의  for (int i = 0; i < n; i++) {  **Widget w(**i에 따라 달라지는 값**)**;             ...  } |

Widget 객체에 들어가는 연산을 기준으로 해서, 2방법에 걸리는 비용을 정리해 보죠.  
그 결과는 다음과 같다.

  A 방법: 생성자1번 + 소멸자1번 + 대입n번

  B 방법: 생성자n번 + 소멸자n번.

클래스 중에는 대입에 들어가는 비용이, 생성자-소멸자 쌍보다 적게 나오는 경우가 있는데, Widget 클래스가 이런 종류에 속한다면 A 방법이 훨씬 효율이 좋습니다. 이 차이는 n이 커질 때 특히 커집니다.  
반면, 그렇지 않은 경우엔 B 방법이 아마 더 좋을 것이고요

A방법을 쓰면 w라는 이름을 볼 수 있는 유효범위가 B방법을 쓸 때보다 넓어지기 때문에(루프를 포함하는 유효범위가 되죠), 프로그램의 이해도와 유지보수성이 역으로 안 좋아질 수도 있습니다.

정리.

1.     대입이 생성자-소멸자 쌍보다 비용이 덜 들고

2.     전체 코드에서 수행 성능에 민감한 부분을 건드리는 중 이라고 생각하지 않는다면, B방법으로 가는 것이 좋습니다.

**요약.**

\* 변수 정의는 늦출 수 있을 때까지 늦춥시다. 프로그램이 더 깔끔해지며 효율도 좋아집니다.