**항목 20. ‘값에 의한 전달’ 보다는 ‘상수 객체 참조자에 의한 전달 방식’을 택하는 편이 대게 낫다.**

기본적으로 C++은 함수로부터 객체를 전달받거나 함수에 객체를 전달할 때 ‘**값에 의한 전달(pass-by-value)’ 방식을 사용** 합니다. (C에서 물려받은 특성 중 하나죠).  
특별히 다른 방식을 지정하지 않는 한, 함수 매개변수는 실제 인자의 ‘사본’을 통해 초기화되며, 어떤 함수를 호출한 쪽은 그 함수가 반환한 값의 ‘사본’을 돌려받습니다. 이들 **사본을 만들어내는 원천이 바로 복사 생성자** 인데요. 이 점 때문에 **‘값에 의한 전달’이 고비용의 연산이** 되기도 합니다.

|  |
| --- |
| class Person {  public:             Person();             virtual ~Person(); // 가상 소멸자인 이유는 항목7에서 확인.             ...    private:             std::string name;             std::string address;  };    class Student : public Person {  public:             Student();             ~Student();             ...    private:             std::string schoolName;             std::string schoolAddress;  }; |

아래에서 validateStudent를 호출하고 있는데, 이 함수는 Student 인자를 전달받고(값으로) 이 인자가 유효화됐는가를 알려 주는 값을 반환받는다.

|  |
| --- |
| bool validateStudent(**Student** s); // Student를 값으로 전달받는 함수.  Student plato;  bool platoIsOK = validateStudent(plato); |

이 함수가 호출될 때 어떤 일이 일어날까요?

plato로부터 매개변수 s를 초기화시키기 위해 Student의 복사 생성자가 호출될 것입니다.  
게다가 s는 validateStudent가 복귀할 때 소멸될 것이고요.  
정리하면, 이 함수의 매개변수 전달 비용은 Student의 복사 생성자 호출 한 번, 그리고 Student의 소멸자 호출 한 번입니다.

여기에 추가로, Student 객체에는 String 객체 2개가 멤버로 들어 있기 때문에,   
Student 객체가 생성될 때마다, 이들 String 형제도 덩달아 생성되어야 합니다.  
게다가 Student 객체는 Person 객체로부터 파생되었기 때문에, Student 객체가 생성되면 Person 객체도 (먼저) 생성되어야 합니다.  
Person 객체 안에는 또 String 객체 2개가 들어 있기 때문에, Person 객체가 매번 생성될 때 String 생성자가 2번 더 불리게 되겠지요.  
**최종 결과는 단지 Student 객체 하나를 값으로 전달했을 뿐인데, Student 복사 생성자 호출 한번, Person 복사 생성자 호출 한 번에 추가로 String 복사 생성자 호출이 4번 일어납니다.**  
Student 객체의 사본이 소멸될 때도 앞서 호출된 생성자들 각각이 소멸자 호출과 대응되는 거죠.

Student 객체를 값으로 전달하는데 생성자 6번, 소멸자 6번의 비용이 듭니다.

생성자 / 소멸자 호출을 몇 번씩 거치지 않고 넘어갈 수 있는 방법.  
**상수 객체에 대한 참조자(reference-to-const)로 전달**하게 만드는 것입니다.

|  |
| --- |
| bool validateStudent(**const** Student**&**s); |

이렇게 하면 순식간에 훨씬 효율적인 코드로 바뀝니다.  
새로 만들어지는 객체 같은 것이 없기 때문에, 생성자와 소멸자가 전혀 호출되지 않거든요.  
원래의 validateStudent는 Student 매개변수를 값으로 받도록 되어 있기 때문에, 호출부에서는 함수로 전달된 Student 객체에 어떤 변화가 생기더라도 그 변화로부터 안전하게 보호를 받는다는 점을 알고 있습니다. 그도 그럴 것이 validateStudent가 상대하는 Student 객체는 원본이 아닌 사본이니까요.  
그런데 이제는 Student 객체의 전달 방식이 참조에 의한 전달입니다. 매개변수 앞에 const가 붙은 건 바로 그 때문인데, 이것이 붙지 않으면 validateStudent함수로 넘어간 Student 객체가 변할지도 모른다는 걱정을 호출부가 해야 하거든요.

참조에 의한 전달 방식으로 매개변수를 넘기면 **복사손실 문제(slicing problem)**가 없어지는 장점도 있습니다.   
파생 클래스 객체가 기본 클래스 객체로서 전달되는 경우는 드물지 않게 접할 수 있는데,   
이때 이 객체가 값으로 전달되면 기본 클래스의 복사 생성자가 호출되고, 파생 클래스 객체로 동작하게 해 주는 특징들이 ‘싹둑 잘려’ 떨어지고 맙니다. 기본 클래스 생성자가 만들었으니까요.

|  |
| --- |
| class Window {  public:             ...             std::string name() const; // 윈도우의 이름을 반환.             virtual void display() const; // 윈도우 테두리 및 내부를 그립니다.  };    class WindowWithScrollBars : public Window {  public:             ...             virtual void display() const;  }; |

어떤 윈도우의 이름을 출력하고, 그 윈도우를 화면에 표시하는 함수를 만든다.  
틀리게 구현한 버전의 예

|  |
| --- |
| void printNameAndDisplay(**Window** w) // 매개변수가 복사손실에 당한다!  {             std::cout << w.name();             w.display();  } |
| 이 함수에 WindowWithScrollBars 객체를 넘긴다면?    WindowWithScrollBars wwsb;  printNameAndDisplay(wwsb); |

매개변수 w가 생성되는데 (값으로 전달됩니다),  
Window객체로 만들어지면서 wwsb가 WindowWithScrollBars 객체의 구실을 할 수 있는 부속정보가 잘려나갑니다.

복사손실 문제를 피하려면, w를 상수객체에 대한 참조자로 전달하도록 만들면 됩니다.

|  |
| --- |
| void printNameAndDisplay(**const** Window**&** **w**) // 매개변수는 잘리지 않습니다.  {             std::cout << w.name();             w.display();  } |

C++컴파일러의 동작 원리에 관심 있는 분이라면, 참조자는 보통 포인터를 써서 구현된다는 사실을 알고 계시거나 알아내실 겁니다.   
즉, **참조자를 전달한다는 것은 결국 포인터를 전달한다는 것과 일맥상통**한다는 이야기죠.  
전달하는 객체의 타입이 기본제공 타입(int 등)일 경우에는 참조자로 넘기는 것보다 값으로 넘기는 편이 더 효율적일 때가 많습니다. 그러니까 ‘값에 의한 전달’ 및 ‘상수객체의 참조에 의한 전달’ 중 하나를 선택해야 할 때, **기본 제공 타입에 대해서는 ‘값에 의한 전달’을 선택하더라도 엉터리가 아니라는 뜻.**이점은 STL의 반복자와 함수 객체에도 마찬가지입니다.  
예전부터 반복자와 함수객체는 값으로 전달되도록 설계해 왔기 때문입니다.   
참고로, 반복자와 함수 객체를 구현할 때는 반드시

1.     복사 효율을 높일 것과

2.     복사손실 문제에 노출되지 않도록 만드는 것

이 필수입니다.

기본제공 타입은 작습니다. 타입 크기만 작으면 전부 ‘값에 의한 전달’을 할 수 있다고 생각할 수 있다.  
‘그냥 크기가 작으니까’는 그 객체의 복사 생성자 호출이 저비용이란 뜻으로 해석하라는 단서가 아닙니다. 이런 객체를 복사하는 데는 그 포인터 멤버가 가리키는 대상까지 복사하는 작업도 따라다녀야 합니다. 크기가 작다고 쉽지 않습니다. 비쌀 수 있다고요.

컴파일러 중에는 기본제공 타입과 사용자 정의 타입을 아예 다르게 취급하는 것들이 있습니다.  
기본제공 타입과 사용자 정의 타입의 하부 표현구조가 같아도 말이죠.  
이를 테면 진짜 double은 레지스터에 넣어주지만, double 하나로만 만들어진 (사용자 정의 타입) 객체는 레지스터에 넣지 않는 것입니다.   
이런 개발 환경에서 일하는 분은 차라리 참조에 의한 전달을 쓰는 편이 좋습니다.

크기가 작다고 해서 작은 사용자 정의 타입을 무조건 값으로 전달할 수 없는 이유가 하나 더 있습니다.   
사용자 정의 타입의 크기는 언제든 변화에 노출되어 있다는 것이죠.  
지금은 크기가 작을지 몰라도 나중에는 커질지도 모르는 노릇입니다.  
개발도구의 내부 구현은 언제든 바뀔 수 있으니까요.

일반적으로, ‘값에 의한 전달’이 저비용이라고 가정해도 괜찮은 유일한 타입은   
기본제공 타입,   
STL 반복자,   
함수 객체 타입,   
이렇게 3가지뿐입니다.

이 외의 타입에 대해서는 ‘상수객체 참조자에 의한 전달’을 선택하세요.

**요약.**

\* ‘값에 의한 전달’보다는 ‘상수 객체 잠조자에 의한 전달’을 선호합시다. 대체적으로 효율적일 뿐만 아니라 복사손실 문제까지 막아줍니다.

\* 이번 항목에서 다룬 법칙은 기본 제공 타입 및 STL 반복자, 그리고 함수 객체 타입에는 맞지 않습니다. 이들에 대해서는 ‘값에 의한 전달’이 더 적절합니다.