**항목 31. 파일 사이의 컴파일 의존성을 최대로 줄이자**

C++의 클래스 정의(class definition)는 클래스 인터페이스만 지정하는 것이 아니라 구현 세부사항까지 상당히 많이 지정한다.

클래스에서 다른 헤더파일을 필요로 하게 된다면 컴파일 의존성이 증가한다.

|  |
| --- |
| int main()  {  int x;  Person p(params);  } |

Person에게 할당된 공간을 알 수 없기에 정의된 정보를 확인할 수 밖에 없다.

인터페이스만 제공하고, 한쪽은 인터페이스의 구현을 맡도록 만드는 것

#include <string> // 표준 라이브러리 구성요소는 전방 선언을 하면 안 됩니다.

표준 라이브러리 헤더는 어지간한 경우만 아니면 컴파일 시 병목 요인이 되진 않습니다.

|  |
| --- |
| #include <memory> // tr1::shared\_ptr을 위한 헤더  class PersonImpl; // Person의 구현 클래스에 대한 전방 선언  class Date; // Person  클래스 안에서 사용하는 것들에 대한 전방 선언  class Address; // Person  클래스 안에서 사용하는 것들에 대한 전방 선언  class Person{  public:  Person(const std::string& name, const Date& birthday, const Address& addr);  std::string name() const;  std::string birthDate() const;  std::string address() const;  private:  std::tr1::shared\_ptr<PersonImpl> pImpl;  }; |

인터페이스와 구현이 분리된다.

(패턴으로 굳어져 있을 정도)

정의부에 대한 의존성을 선언부에 대한 의존성으로 바꾸어 놓는다.

이게 바로 컴파일 의존성을 최소화 하는 핵심 원리.

**- 객체 참조자 및 포인터로 충분한 경우에는 객체를 직접 쓰지 않는다.**

  # 어떤 타임에 대한 참조자 및 포인터를 정의할 때는 그 타입의 선언부만 필요하다.

    반면, 어떤 타입의 객체를 정의할 때는 그 타입의 정의가 준비되어 있어야 합니다.

**- 할 수 있으면 클래스 정의 대신 클래스 선언에 최대한 의존하도록 만들자**

  # 어떤 클래스를 사용하는 함수를 선언할 때는 그 클래스의 정의를 가져오지 않아도 됩니다. 심지어 그 클래스 객체를 값으로 전달하거나 반환하더라도 클래스 정의가 필요 없다.

|  |
| --- |
| class Date; // 클래스 선언  Date today();                   // Date 클래스의 정의를 가져오지 않아도 됩니다.  void clearAppointments(Date d); |

물론 값에 의한 전달 방식은 좋은 방법이라고 보긴 힘들지만 피치 못할 사정 떄문에 써야 할 경우도 있을 텐데, 이런 경우에도 불필요한 컴파일 의존성을 끌고 들어오는게 아무런 변명거리도되지 못한다는 것.

**- 선언부와 정의부에 대해 별도의 헤더 파일을 제공합니다.**

제품을 만들려면 클래스 정의를 제공하는 일을 어딘가에서 해야 하겠지만, 함수 선언이 되어 있는 여러분의 헤더파일쪽에 부담을 주지않고 실제 함수 호출이 일어나는 사용자의 소스 파일 쪽에 전가하는 방법을 사용한 것입니다.

"클래스를 둘로 쪼개자"라는 지침을 제대로 쓸 수 있도록 하려면 헤더 파일이 짝으로 있어야 합니다.

하나는 선언부를 위한 헤더 파일이고, 또 하나는 정의부를 위한 헤더 파일이겠지요.

이 두 파일은 관리도 짝 단위로 해야합니다. 한쪽에서 어떤 선언이 바뀌면 다른 쪽도 똑같이 바꾸어야 한다는 것

그렇기 때문에 라이브러리 사용자 쪽에서는 전방 선언 대신에 선언부 헤더 파일을 항상 #include해야 할 것이고, 라이브러리 제작자 쪽에서는 헤더 파일 두 개를 짝지어 제공하는 일을 잊으면 안 됩니다.

**C++에서는 템플릿 선언과 템플릿 정의를 분리할 수 있도록 하는 기능을 export라는 키워드로 제공하고 있습니다.**

**pImpl 관용구를 사용하는 Person같은 클래스를 가리켜 핸들 클래스라고 합니다.**

핸들 클래스에서 어떤 함수를 호출하게 되어 있다면, 핸들 클래스에 대응되는 구현 클래스 쪽으로 그 함수 호출을 전달해서 구현 클래스가 실제 작업을 수행하게 만드세요.

|  |
| --- |
| #include "Person.h" // Person클래스를 구현하고 있는 중이기 때문에 Person의 클래스 정의를 include해야 합니다. |

#include "PersonImpl.h" // 이와 동시에 PersonImpl의 클래스 정의도 include해야 하는데, 이렇게 하지 않으면 멤버 함수를 호출할 수 없습니다. PersonImpl 의 멤버함수는 Person의 멤버 함수와 일대일로 대응되고 있음을 알 수 있습니다.

|  |
| --- |
| Person::Person(const std::string& name, const Date& birthday, const Address& addr)  : pImpl(new PersonImpl(name,birthday, addr))  {}  std:: string Person::name() const  {  return pImpl->name();  }  이 방법이 아니라면 Person을 특수 형태의 추상 기본 클래스,이른바 인터페이스 클래스로 만드는 방법을 생각해 볼 수 있다.  class Person{  public:  virtual ~Person();  virtual std::string name() const = 0;  virtual std::string birthDate() const = 0;  virtual std::string address() const = 0;  ...  }; |

이 클래스를 코드에 써먹으려면 Person에 대한 포인터 혹은 참조자로 프로그래밍하는 방법밖에 없습니다.

인터페이스 클래스를 사용하기 위해서는 객체 생성 수단이 최소한 하나는 있어야 합니다.

주어진 인터페이스 클래스의 인터페이스를 지원하는 객체를 동적으로 할당한 후, 그 객체의 포인터를 반환하는 것

(팩토리 함수 혹은 가상 생성자)

핸들 클래스를 사용하게 되면 구현부의 객체의 데이터까지 가기위한 포인터를 타야함으로서 요구되는 간접화 연산이 하나 증가됨

객체 하나씩을 저장하는데 필요한 메모리 크기에 구현부 포인터의 크기가 더해짐

구현부 포인터가 동적 할당된 구현부 객체를 가리키도록 어디선가 그 구현부 포인터의 초기화가 일어나야함.

동적 메모리 할당에 따르는 연산 오버헤드, bad\_alloc 예외와 맞부딪힐 가능성 까지 있다.

인터페이스 클래스를 사용하는 경우에는 전부 가상함수라는 것이 약점

함수호출이 일어날 떄마다 가상 테이블 점프에 따르는 비용이 소모된다.(항목 7)

가상 함수를 공급하는 쪽이 인터페이스 클래스 밖에 없을때는, 이 가상 테이블 포인터도 객체 하나를 저장하는데 필요한 메모리 크기를 늘리는 요인이 된다.

인터페이스 클래스와 핸들클래스 모두 가지고 있는 약점은 인라인 함수의 도움을 제대로 끌어내 힘들다는 점.

인라인이 되게 만들려면 함수 본문을 대개 헤더 파일에 두어야 하는데 핸들클래스와 인터페이스 클래스는 함수 본문과 구현부를 사용자의 눈으로부터 차단하므로 인라인과 친해질 수가 없다.

\*요약

- 컴파일 의존성을 최소화 하는 작업의 배경이 되는 가장 기본적인 아이디어는 ‘정의’대신에 ‘선언’에 의존하게 만들자는 것입니다. 이 아이디어에 기반한 두 가지 접근 방법은 핸들 클래스와 인터페이스 클래스입니다.

- 라이브러리 헤더는 그 자체로 모든 것을 갖추어야 하며 선언부만 갖고 있는 형태여야 합니다. 이 규칙은 템플릿이 쓰이거나 쓰이지 않거나 동일하게 적용합시다.