객체 지향 프로그래밍
(Object Oriented Programming)

14장

강사 – 강대기

차례 (Agenda)

- has-a 관계
- 객체 멤버를 가지는 클래스 (컨테인먼트)
- valarray 템플릿 클래스
- private 상속과 protected 상속
- 다중 상속
- 가상 기초 클래스
- 클래스 템플릿 만들기
- 클래스 템플릿 사용하기
- 템플릿 특수화

코드의 재활용성

- 상속(inheritance) 대 합성(composition)
- 상속
  - 인터페이스 상속 (interface inheritance)
    - is-a 관계
    - C++ 에서는 public 상속
  - 구현 상속 (implementation inheritance) 또는 클래스 상속
    - has-a 관계
    - C++ 에서는 private 상속, protected 상속
- 합성
  - 컴포지션(composition), 컨테인먼트(containment), 레이어링(layering)
  - has-a 관계
- has-a 관계는 컨테인먼트 또는 private 상속

Student class 의 예

- 이름
  - (1) 문자 배열, (2) 포인터와 동적 메모리 할당, (3) string 클래스
- 성적
  - (1) 고정 크기 배열, (2) 동적 메모리 할당, (3) valarray 클래스
- valarray 클래스 (891쪽) ← 클래스 템플릿
  - valarray<int> q_values; // 크기 0
  - valarray<int> v2(8); // 크기 8
  - valarray<int> v3(10,8); // 크기 8, 초기값 10
  - valarray<double> v4(gpa,4); // 크기 4, 초기값 gpa 원소
  - operator[](), size(), sum(), max(), min()

컨테인먼트: Student 클래스 설계 (892쪽)

```
class Student {
private:
  string name;
  valarray<double> scores;
}
```

- 구현을 획득하지만, 인터페이스는 상속하지 않음! (중요) → 구현 획득

컨테인먼트: Student 클래스 예제 (894쪽)

- typedef – 클래스 정의의 private 부분
- explicit 키워드 (688쪽)
- 제한을 가하는 도구들은 적극적으로 활용 (896쪽)
- 멤버 초기자 리스트
  - 객체 멤버는 상속의 경우 기초 클래스와 마찬가지로 미리 생성되어 초기화되어야 함
  - 따라서, 사용하지 않을 경우 다른 멤버 객체들은 디폴트 생성자로 생성됨
  - 초기화 순서에 조심한다
- 클래스 내부에선 내포된 객체의 메써드를 사용할 수 있음 → 필요하면 헬퍼 메쏘드를 사용 (898쪽)

private 상속 (903쪽)

- private 상속은 디폴트 상속 방법임
- public 상속 – 인터페이스 상속
- private 상속 – 구현 상속
- 컨테인먼트 – 종속 객체(subobject)의 구현획득

private 상속: Student 클래스

- 다중 상속 (904쪽)
- private 멤버 데이터를 가질 필요가 없음 – 상속되는 두 기초 클래스가 필요한 데이터 멤버들을 제공
- 기초 클래스 성분의 초기화
  - 멤버 이름이 아닌 클래스 이름을 사용(905쪽)
- 기초 클래스 메쏘드에 접근
  - 클래스 이름과 클래스 사용 범위 연산자 사용 (907쪽)
    1. ArrayDB::sum()
    2. std::valarray<double>::sum()
- 기초 클래스 객체에 접근 (907쪽) – 위의 3,4,5
- 기초 클래스 프렌드에 접근 (908쪽) – 위의 3,4,5
- private 상속에서는 파생 클래스에서 기초 클래스로의 명시적 데이터 형 변환만 가능함 (암시적 데이터 형 변환은 안됨)
  - public 상속이라도 명시적 데이터 형 변환 필요 (909쪽 참고)

컨테인먼트와 private 상속 비교

- Favor object composition over class inheritance! – GoF
  1. 컨테인먼트가 더 사용하기 쉽다
  2. 다중 상속은 다이아몬드 구조가 되는 경우 문제를 일으킬 소지가 큼
     1. 두 개 이상의 기초 클래스들이 같은 조상을 공유할 경우, 파생 클래스가 그 조상의 다중 인스턴스를 가짐
     2. 두 개 이상의 기초 클래스들이 같은 이름을 가진 메쏘드들을 공유
  3. 같은 클래스 종속 객체들을 여러 개 내포할 수 있음
- private 상속의 (허접한) 장점 (컨테인먼트에 대한)
  1. protected 멤버를 사용할 수 있음
  2. 가상 멤버 함수를 다시 정의할 수 있음
- protected 상속 – private 상속의 변종
  - 915쪽의 <표 14.1>

using을 사용하여 접근을 다시 정의

- private/protected 상속을 하고 나서, 기초 클래스들의 특정 메쏘드들을 public으로 하려면?
  1. public 파생 클래스 메쏘드를 새로 정의
  2. using 선언 사용 (916쪽)
  3. 구식 방법: public으로 다시 선언
     – 원래의 의도를 퇴색시킴

다중 상속

- 문제점
  1. 서로 다른 두 기초 클래스들에서 이름은 같지만 서로 다른 메쏘드를 상속하는 문제
  2. 같은 조상 클래스를 가지는 서로 다른 두 기초 클래스들에게서 다중 인스턴스를 상속받는 문제
- 다중 상속을 반대하는 사람들이 더 많음
- 다중 상속이 들어 맞는 경우
  - 템플릿
  - Policy-based programming
- Java – 클래스의 다중 상속 안됨
  - Java에서는 어떻게 다중 상속을 흉내낼 수 있는가? (시험문제)
- 클래스 구조 – Worker(조부모), Waiter(부모), Singer(부모), SingingWaiter(자식)

조부모 클래스의 다중 인스턴스(923쪽)

- 두 개의 Worker 성분 – 924쪽의 표 14.4
- 모호성 – (923쪽)

```
SingingWaiter ed;
Worker* pw = &ed;
```

  – 강제 데이터 형 변환으로 해결 → 다형성을 해침

- 해결책은 가상 기초 클래스

```
class Singer : virtual public Worker {};
class Waiter : virtual public Worker {};
```

가상 기초 클래스 (924쪽)

- 하나의 공통 조상을 공유하는 여러 개의 기초 클래스로부터 공통 조상의 유일 객체를 상속 받는 방식
- 조상 객체는 하나만 가짐
  - 그림 14.4 (924쪽) 및 그림 14.5 (925쪽) 참조
- 이 시점에서 의문점들
  1. 왜 가상이냐?
  2. 처음부터 디폴트로 가상이면 안되냐?
  3. 가상 기초 클래스를 쓰려면 뭘 해야 하나?

가상 기초 클래스 의문점 해결!

- 왜 가상이라고 하냐?
  - 별 의미 없음.
  - 새로운 키워드를 추가하기 힘들다
  - 일종의 키워드 오버로딩!
- 왜 가상 기초 클래스가 디폴트가 아닌가?
  - 여러 벌의 복사본을 원하는 경우가 있음
  - 가상 기초가 디폴트이려면 추가 작업이 필요함
  - 모호성
    - 조상 클래스의 생성자를 명시적으로 호출해야 함
    - 메쏘드 사용의 모호성이 존재함

가상 기초 클래스 의문점 해결!

- 가상 기초 클래스를 쓰려면 뭐를 해야 하나?
  - virtual 키워드 추가
  - 기존 코드들의 변경

가상 기초 클래스의 생성자 규칙

SingingWaiter (const Worker & wk,...) :
  Waiter(wk,p), Singer(wk,v) {}

- 정보의 전달이 두 개의 다른 경로를 거친다!
- 가상이 아닌 경우, 각각의 카피로 가지만, 가상이라면 문제가 됨
- 해결책: 명시적으로 조상 생성자도 호출

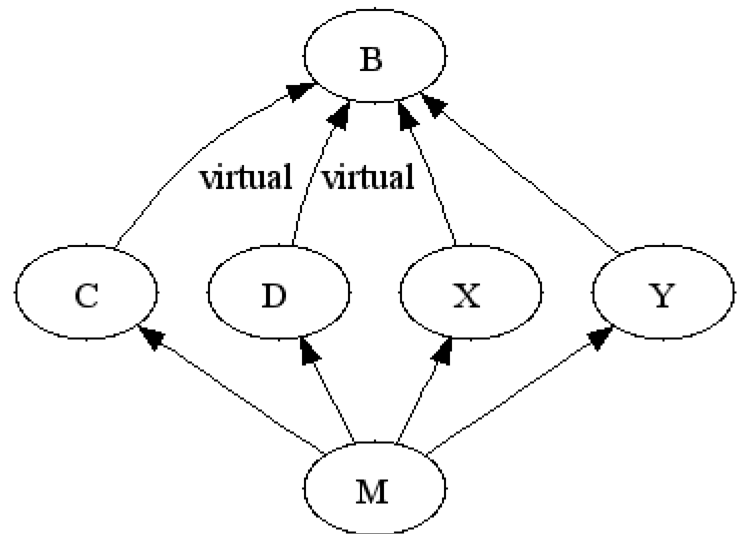
SingingWaiter (const Worker & wk,...) :
  Worker(wk), Waiter(wk,p), Singer(wk,v) {}

메쏘드 사용의 모호성 (928쪽)

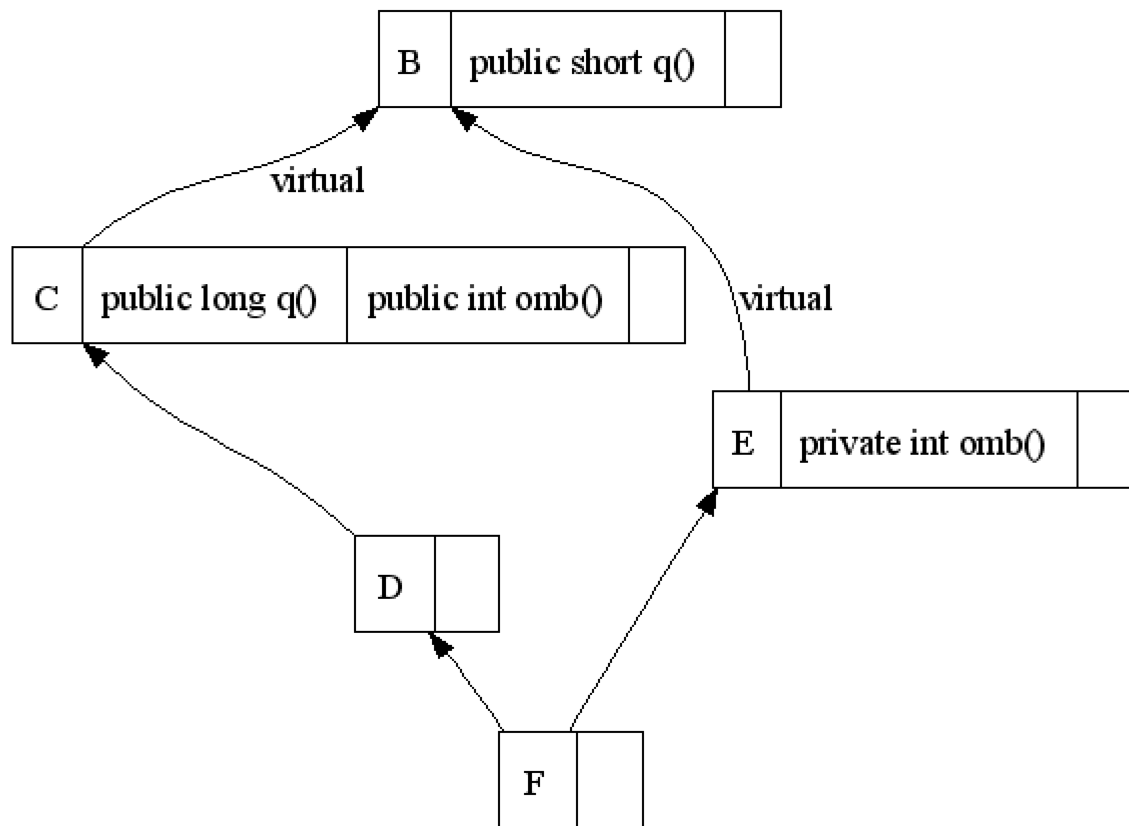
- SingingWaiter newhire(...);
- newhire.Show(); // 모호함
- newhire.Singer::Show(); // 해결
- 점층적 접근 방식은 문제가 됨 (929쪽)
  - → private 헬퍼 메쏘드를 통한 모듈 접근 방식을 사용 (930쪽)

가상 기초 클래스와 가상이 아닌 기초 클래스의 혼합

- 다음과 같은 경우
  - 클래스 M은 C,D에 대해서는 클래스 B의 종속 객체 하나만 내포
  - 클래스 M은 X,Y에 대해 각각 별개의 클래스 B의 종속 객체들을 내포
- 모든 가상 경로에 대해서는 하나의 클래스 객체, 그렇지 않으면 별개의 기초 클래스 종속 객체들을 내포함



가상 기초 클래스의 비교 우위 (dominance) (참고: 842쪽 오버로딩 대 오버라이딩)



- 가상이 아닌 클래스의 경우 모호함 → 가상 클래스인 경우 모호할 수도 있고, 비교 우위에 의해 모호하지 않을 수도 있음
- 파생 클래스의 이름은 조상 클래스보다 비교 우위를 가짐

가상 기초 클래스의 비교 우위 (dominance)

- 클래스 C가 B에서 파생되었으므로, 클래스 C의 q() 정의가 클래스 B의 q() 정의보다 비교 우위를 가짐 → F의 메쏘드들은 C::q()를 q()라고 할 수 있음
- C와 E는 서로 상대에 대해 기초 클래스가 아니므로, omb() 정의들은 서로 비교 우위를 가지지 않음 → F에서 omb()를 하면 모호해짐
- 가상 모호성은 접근 규칙은 고려하지 않음 → E의 omb() 가 private 이라고 해도 여전히 omb()의 사용은 모호함

클래스 템플릿 (942쪽)

- 예를 들면 데이터 형과 무관한 형식으로 스택을 정의할 수 있는가?
- typedef? – 943쪽
  - typedef unsigned long Item;
  - 단점 1 – 데이터 변경할 때마다 헤더 파일 수정
  - 단점 2 – 프로그램 당, 한 종류의 스택만 생성
- C++의 해결책 – 클래스 템플릿

일단 함수 템플릿 복습 (466쪽)

- 템플릿은 함수가 아니며 무조건 인라인 함수가 되는 것도 아님
- Swap 함수의 템플릿 예

```
template <typename T>
void Swap (T &a, T &b)
{
  T temp = a;
  a =b;
  b = temp;
}
```

- 위의 선언을 했다고 해도 아직 함수가 생긴 게 아님

함수 템플릿 복습 계속

- 특수화 (specialization)
  - 템플릿은 추상이다. 따라서 코드로 실행하려면 특수한 예를 이끌어 내야 한다.
    - 암시적 구체화, 명시적 구체화, 명시적 특수화
  - 구체화 또는 인스턴스화 (instantiation) 또는 암시적 특수화
    - 템플릿은 함수가 아니다. 템플릿으로부터 함수를 만들어 내야 한다.
    - 암시적 구체화 – 마치 있었던 거처럼 자연스럽게 요구함
    - 명시적 구체화 – 대놓고 요청함. 라이브러리 만들 때 좋음
  - 명시적 특수화 (explicit specialization)
    - 템플릿에서 지정한 방법과는 다른 특수한 예외를 만들고 싶다

함수 템플릿 복습 계속

- 암시적 구체화 (implicit instantiation)
  - 코드를 만들라는 명시적인 선언이나 실행이 없다. 다만 이미 만들었겠거니 하고 내부적으로 돌아갈 뿐 (468쪽과 471쪽 코드)
  - 469쪽의 Swap(i,j) 는 Swap<int>(i,j) 와 동일함
- 명시적 구체화 (explicit instantiation)
  - 프로그램 내부에서 선언해서 구체적인 코드를 만들라고 명한다. (479쪽)
  - 예: template void Swap<int>(int &, int &);
- 명시적 특수화 (explicit specialization)
  - 템플리트의 특수한 경우를 만들고자 한다 (475쪽 코드)
  - 예: template <> void Swap<int>(int &, int &);
  - 예: template <> void Swap(int&, int&);

함수 템플릿 복습 계속

- 템플릿도 오버로딩된다 (471쪽)
- 따라서 함수 이름이 하나 주어지면 (474쪽)
  - 템플릿이 아닌 그냥 함수, 템플릿 함수 (실은 암시적 특수화 함수), 명시적 특수화 함수
  - 그리고 이것들의 오버로딩 버전들이 존재할 수 있음
- 가장 적합한 버전을 찾는 것을
  - 오버로딩 분석(overloading resolution)이라고 함
  - 일반적으로 가장 특수화된 경우를 찾음

클래스 템플릿

- 예를 들면 데이터 형과 무관한 형식으로 스택을 정의 할 수 있는가?
- 템플릿 – 매개변수화되는 데이터형 제공
- 예: Queue에 int를 전달하여 int의 Queue 생성, valarray 템플릿 클래스
- 클래스 템플릿 정의 (945쪽)
  - template <typename T> 로 시작
  - 데이터 형 → T
  - template <typename T> bool Stack<T>::push(const T& item)
- 템플릿 선언과 함수를 분리시키고 싶다면, export

포인터들의 스택 템플릿

- Stack<char *> st 를 적용하는 경우, 잘못된 응용 (951쪽)
  - string po; → char* po;
    - 포인터만 있지 저장 공간 없음 → 실행 시간 에러
  - string po; → char po[40];
    - pop 메쏘드와 안맞음 → 배열 이름에 대입 안되므로 컴파일 에러
  - string po; → char* po = new char[40];
    - po 주소는 바뀌지 않고 스택에 들어감 → 로직 에러
- 올바른 사용 → 호출한 프로그램이 포인터들의 배열을 제공하는 것

포인터들의 스택 템플릿

- 원래 스택 – 945쪽
- 바뀐 스택 (953쪽)
  - 동적 메모리 할당으로 포인터들의 배열을 할당함
- 제대로 된 사용 예 – 956쪽

배열 템플릿 예제와 데이터 형이 아닌 전달 인자

- 컨테이너 클래스 – 객체들을 담기 위한 클래스
  - 템플릿은 다양한 데이터 형이 매개 변수로 사용된다는 점에서 컨테이너 클래스와 잘 맞아 떨어짐
  - 컨테이너 클래스들을 위해 재활용하기 위한 코드를 만들자는 게 템플릿을 도입한 동기
- 배열 템플릿 예제 – 958쪽
  - template <class T, int n>
  - T는 데이터 형, n은 숫자 (수식 전달 인자)
  - ArrayTP<double, 12> eggweights;
- 수식 전달 인자double 형은 허용 안됨

수식 전달 인자

- 정수형, 열거형, 참조, 포인터 허용됨
  - double& rm, double* pm
- double 형은 허용 안됨
  - double m; // 안됨
- 템플릿 코드는 수식 전달 인자 값을 변경하거나 그 주소를 얻을 수는 없음
  - n++, &n 등은 허용 안됨
  - 템플릿을 구체화할 때, 수식 전달 인자의 값은 상수이어야 함
- 생성자 접근 방식 – 힙 메모리
- 수식 전달 인자 – 스택 – 크기가 작은 배열이면 더 빠름

수식 전달 인자

- 가장 큰 단점
  - 각 배열 크기가 자신만의 템플릿을 각각 생성
  - ArrayTP<double, 12> eggweights;
  - ArrayTP<double, 13> donuts;
  - Stack<int> eggs(12);
  - Stack<int> dunkers(13);
- 생성자 접근 방식은 배열 크기를 클래스 멤버로 저장하므로 더 융통성이 있음

템플릿의 융통성과 재귀적 사용

- 템플릿은
  - 기초 클래스가 될 수도 있고
  - 성분 클래스가 될 수도 있고
  - 다른 템플릿의 데이터형 매개 변수가 될 수도 있다
- 배열 템플릿으로 배열 원소들이 하나의 스택 템플릿을 이루는 스택 템플릿을 만들고, 다시 배열 템플릿으로 스택 템플릿의 배열 생성(961쪽)
  - Array < Stack<int> > asi; // > >로 >>와 구별
- 템플릿의 재귀적 사용 – 962 쪽
  - ArrayTP< ArrayTP<int, 5> , 10 > twodee;
  - int twodee[10][5]; 와 동일

하나 이상의 데이터 형 매개 변수, 그리고 디폴트 데이터 형 매개 변수

- 하나 이상의 데이터 형 매개 변수 – 962 쪽
  - Pair<string, int>
- 디폴트 데이터 형 매개 변수
  - template<class T1, class T2=int> class Topo …
  - Topo<double, double> m1;
  - Topo<double> m2; // Topo<double, int> m2;
  - 클래스 템플릿 데이터 형 매개 변수 – 디폴트 값 가능
  - 함수 템플릿 데이터 형 매개 변수 – 디폴트 값 불가능
  - 데이터 형 아닌 매개 변수 – 디폴트 값 가능

템플릿 특수화

- 암시적 구체화
  - 사용하기를 원하는 데이터 형을 나타내는 객체를 선언 (예: ArrayTP<int,100> stuff;)
  - 객체가 요구될 때까지 암시적 구체화를 미룸
    - ArrayTP<double, 30> *pt; // 아직 아님
    - pt = new ArrayTP<double, 30>; // 암시적 구체화!
- 명시적 구체화
  - 클래스 선언
    - template class ArrayTP<string, 100>;
- 명시적 특수화 – 다음 슬라이드

명시적 특수화

- 템플릿이 특정 형에 맞게 구체화될 때, 조금 다르게 행동하도록 수정
  - template<class T> class SortedArray { … }
  - T::operator>() 메소드는 T가 char* 인 경우 다르게 동작함
  - template<> class SortedArray<char*> {…}
- 부분적인 특수화
  - 템플릿의 포괄성을 일부 제한하는 것을 말함
  - 예
    - 데이터 형 매개 변수 중 어느 하나에 구체적 데이터 형 제공
    - 포인터들을 위한 특별한 버전 제공

부분적인 특수화

- 데이터 형 매개 변수 중 어느 하나에 구체적 데이터 형 제공
  - 포괄적인 경우
    - template<class T1, class T2> class Pair {…};
  - 부분적인 특수화
    - template<class T1> class Pair<T1, int> {…};
  - 명시적 특수화
    - template<> class Pair<int, int> {…};
- 사용 예
  - Pair <double, double> p1; // 포괄적인 경우
  - Pair <double, int> p2; // 부분적인 특수화
  - Pair <int, int> p3; // 명시적 특수화

부분적인 특수화

- 포인터들을 위한 특별한 버전 제공
  - 포괄적인 버전
    - template<class T> class Feeb {…};
  - 부분적인 특수화
    - template<class T*> class Feeb {…};
- 사용 예
  - Feeb<char> fb1; // 포괄적 버전, T는 char
  - Feeb<char *> fb2; // 부분적 특수화, T는 char

부분적인 특수화

- 다양한 제한 허용
  - 포괄적인 버전
    - template<class T1,class T2, class T3> class Trio {…};
  - T3를 T2로 설정하는 특수화
    - template<class T1,class T2> class Trio<T1,T2,T2> {…};
  - T3와 T2를 T1*로 설정하는 특수화
    - template<class T1> class Trio<T1,T1*,T1*> {…};
- 사용 예
  - Trio<int, short, char *> t1; // 포괄적 버전
  - Trio<int, short> t2; // Trio<T1,T2,T2>
  - Trio<char,char*,char*> t3; // Trio<T1,T1*,T1*>

멤버 템플릿

- 템플릿은 구조체, 클래스, 템플릿 클래스의 멤버가 될 수 있음 – STL에서 요구됨
- 970쪽 예제
  - 내포된 템플릿 클래스 멤버

매개변수 템플릿

- 템플릿의 매개 변수는
  - 데이터 형 매개 변수, 데이터 형이 아닌 매개 변수, 그리고 템플릿 매개 변수 를 가질 수 있음
- 974쪽 예제
  - template <template <typename T> class Thing> class Crab {…};
  - Crab<King> legs; 를 선언하면
    - Thing<int>가 King<int>로
    - Thing<double>이 King<double>로 바뀜
  - Crab<Stack> nebula; 를 선언하면
    - Thing<int>가 Stack<int>로
    - Thing<double>이 Stack<double>로 바뀜

매개변수 템플릿

- 일반 매개 변수와 혼합 가능
- 사용 예
  - template <template <typename T> class Thing, typename U, typename V> class Crab {…};
  - Crab<Stack, int, double> nebula;

템플릿 클래스와 프렌드 함수

- 템플릿의 프렌드는 세가지가 있음
  - 템플릿이 아닌 프렌드
  - 바운드 템플릿 프렌드 – 클래스가 구체화될 때, 클래스의 데이터 형에 의해 프렌드의 데이터 형이 결정됨
  - 언바운드 템플릿 프렌드 – 프렌드의 모든 특수화가 그 클래스의 각 특수화에 대해 프렌드들임

템플릿이 아닌 프렌드 함수

```
template <class T>
class HasFriend
{
  friend void counts(); // 모든 HasFriend 구체화에 대한 프렌드 함수
}
void counts() {…}
```

- counts()는 HasFriend<int> 클래스에 대해서도 프렌드이고, HasFriend<double> 클래스에 대해서도 프렌드임

바운드 템플릿 프렌드 함수

```
template <class T>
class HasFriend
{
friend void report(HasFriend<T> &); // 바운드 템플릿 프렌드 함수
}
void report(HasFriend<int> &hf) {…}
void report(HasFriend<double> &hf) {…}
```

- 위의 두 report는 HasFriend의 각각의 특수화에 대한 프렌드로, 사용되는 모든 특수화에 대해 정의해 주어야 함

템플릿 클래스의 특수화에 따라 자동으로 프렌드 함수가 나오게 하는 법

- 프렌드 함수들을 함수 템플릿으로 만들어 바운드 템플릿 프렌드로 설정하면 클래스 템플릿과 일체화가 됨 (981 쪽)

1. 클래스 정의 앞에 템플릿 함수 선언
   - template <typename T> void counts();
   - template <typename T> void report(T &);
2. 템플릿 안에서 함수 템플릿들을 프렌드로 선언
   - friend void counts<TT>();
   - friend void report<>(HasFriendT<TT> &);
     - or friend void report< HasFriendT<TT> >(HasFriendT<TT> &);
3. 그 프렌드 선언에 대한 함수 템플릿 정의 제공
4. 사용 예
   - counts<int>(); counts(double)();
   - report(hfi2); // 또는 report< HasFriendT<int> >(hfi2);

언바운드 템플릿 프렌드 함수

- 프렌드 템플릿 데이터 형 매개 변수들이 템플릿 클래스 데이터 형 매개 변수들과 다름 (984쪽)

```
template <typename T>
class ManyFriend
{
  template <typename C, typename D> friend void
   show2(C&, D&);
}
```

- 사용 예
  - show2(hfi1, hfi2);
    - show2<ManyFriend<int>, ManyFriend<int> > (hfi1,hfi2);
  - show2(hfdb, hfi2);
    - show2<ManyFriend<double>, ManyFriend<int>> (hfdb, hfi2);