제16장

템플릿

학습 목표

- 함수 템플릿
  - 구분, 정의
  - 컴파일러의 복잡성
- 클래스 템플릿
  - 구문
  - 예제: 배열 템플릿 클래스
- 템플릿과 상속
  - 예제: 부분적으로 채워진 배열 템플릿 클래스

소개

- C++ 템플릿
  - Java의 generic에 대응
  - 함수와 클래스에 대해 매우 일반적인 정의가 가능
  - 형 이름은 실제 유형 대신에 매개변수라 함
  - 실행 중에 정확한 정의가 결정

함수 템플릿

- swapValues 함수:
  - 오직 지정된 자료형의 변수에만 적용

```
void swapValues(int& var1, int& var2)
{
    int temp;
    temp = var1;
    var1 = var2;
    var2 = temp;
}
```

```
void swapValues(char& var1, char& var2)
{
    char temp;
    temp = var1;
    var1 = var2;
    var2 = temp;
}
```

- template swapValues 함수
  - 첫 번째 라인은 "템플릿 전위문"
    - 뒤따라 나오는 것들이 "template" 임을 컴파일러에게 말하는 것
  - T는 형 매개변수
    - 다른 값을 쓸 수 있지만, 주로 T를 사용

```
template<class T>
void swapValues(T& var1, T& var2)
{
    T temp;
    temp = var1;
    var1 = var2;
    var2 = temp;
}
```

함수 템플릿 정의와 호출

- swapValues() 함수 템플릿은 실질적으로 정의의 큰 집합!
  - 각 가능한 유형의 정의!
- 컴파일러는 오직 요구될 때 정의를 자동 생성시킴
  - 그러나 마치 모든 유형에 대해 정의된 것처럼 보이지만
- 하나의 정의를 만들면 → 요구된 모두 유형에 동작

- 호출 문장 : `swapValues(int1, int2);`
  - 컴파일러가 템플릿을 사용하는 2개의 int 매개변수를 위한 함수 정의 생성
  - 비슷하게 모든 유형에 대해서도

다른 함수 템플릿의 예

```
template<class T>
void showStuff(int stuff1, T stuff2, T stuff3)
{
     cout << stuff1 << endl
          << stuff2 << endl
          << stuff3 << endl;
}
```

- 다음 함수 호출을 살펴보자: `showStuff(2, 3.3, 4.4);`
- 컴파일러는 함수 정의 생성
  - T 를 double로 대체
    - 2번째 매개변수부터 double형임
- 결과

```
2
3.3
4.4
```

컴파일러 복잡성

- 함수 선언과 정의
  - 전형적으로 분리해서 사용
  - 템플릿에서 → 대부분의 컴파일러가 지원하지 않음!
- 함수 템플릿 정의를 템플릿 함수를 호출하는 파일에 같이 두는 것이 가장 안전
  - 많은 컴파일러가 처음에 나타나기를 요구함
  - 종종 모든 템플릿 정의를 #include
- 사용하는 컴파일러가 요구하는 사항을 확인해야 함
  - 몇몇 컴파일러는 특별한 옵션 설정을 요구
  - 몇몇은 다른 파일 아이템들에 비해 템플릿 정의의 특별한 순서를 요구
- 대부분 이용가능한 템플릿 프로그램 레이아웃:
  - 템플릿 정의는 그것을 사용하는 파일 안에
  - 어떠한 사용보다도 앞서도록 #include 사용 가능

여러 개의 형 매개변수

- 다음 가능: template<class T1, class T2>
- 비전형적
  - 보통 하나의 대체 가능한 유형만 필요
  - 사용하지 않는 템플릿 매개변수를 가져서는 안됨
    - 각각은 정의에서 사용되어야만 함
    - 그렇지 않으면 에러!

알고리즘 추상화

- 템플릿을 구현하는 것을 나타냄
- 일반적인 알고리즘을 표현:
  - 알고리즘은 어떠한 유형의 변수에 적용
  - 부수적으로 따르는 자세한 것을 무시
  - 알고리즘의 중요한 부분에만 집중
- 함수 템플릿은 알고리즘 추상화를 지원하는 C++의 특징 중 하나

- 템플릿 정의 전략
  - 실제 데이터 형을 사용하여 일반 함수를 개발하고 디버그 완료
  - 그리고 나서 원하는 형 이름을 형 매개변수로 대체하여 템플릿으로 변환
  - 장점:
    - 확실한 경우를 해결하기 쉬워짐
    - 템플릿 구문 규칙이 아닌 알고리즘을 다룸

템플릿에서 부적절한 유형

- ADT가 적절히 구현되었다면 템플릿에서 사용 가능
- 예, swapValues() 템플릿 함수
  - 정의되지 않은 할당 연산자에 대해 유형을 사용할 수 없음
  - 예제: 배열:

```
int a[10], b[10];
swapValues(a, b);
```

    - 정적 배열은 대입이 불가능 하므로, 컴파일 오류 발생

클래스 템플릿

- 마찬가지로 클래스를 일반화 가능 클래스 정의에 적용 가능
  - 클래스 정의에서 모든 "T" 의 인스턴스는 형 매개변수에 의해 대체된다.
  - 함수 템플릿과 같이!
- 한번 템플릿이 정의되면, 클래스의 객체를 선언 가능

```
template<class T>
class Pair
{
public:
    Pair();
    Pair(T firstVal, T secondVal);
    void setFirst(T newVal);
    void setSecond(T newVal);
    T getFirst() const;
    T getSecond() const;
private:
    T first; T second;
};
```

```
template<class T>
Pair<T>::Pair(T firstVal, T secondVal)
{
    first = firstVal;  second = secondVal;
}
template<class T>
void Pair<T>::setFirst(T newVal)
{
    first = newVal;
}
```

```
Pair<int> score;
Pair<char> seats;

score.setFirst(3);
score.setSecond(0);
```

매개변수로서 클래스 템플릿

- 살펴보자: `int addUP(const Pair<int>& the Pair);`
  - 이 클래스 형 매개변수 정의에서 T형 자리에 채워질 형은 int형
  - 참조에 의한 호출이 일어난다
- Again: 템플릿 형은 기본형이 사용되는 어디에든 사용될 수 있다

- 함수 template안에서 class template

```
template<class T>
T addUp(const Pair<T>& the Pair);
      //선행조건: 1 연산자가 T형의 값에 대해 정의되어 있다.
      //thePair에 있는 값 2개의 합을 리턴한다.
```

형 매개변수의 제약사항

- 오직 합리적 유형이 T에 대체될 수 있음
- Consider:
  - 할당 연산자가 잘 작동해야만 함
  - 복사 생성자 또한 잘 작동해야 함
  - 만약 T가 포인터를 포함하면, 적절한 소멸자도 가져야 함!
- 이와 같은 것은 함수 템플릿도 비슷함

형 정의

- 새로운 클래스 형 이름을 정의 가능
  - 특수화된 클래스 이름을 표현하기 위한
- 예: `typedef Pair<int> PairOfInt;`
  - "PairOfInt" 이름은Pair<int> 형의 객체 선언에 사용됨: `PairOfInt pair1, pair2;`
    - 형식 매개변수의 형 또는 형 이름이 올 수 있는 어디든지 사용될 수 있음

프렌드와 템플릿

- 프렌드 함수는 템플릿 클래스에 사용될 수 있음
  - 일반 클래스와 같은 방식으로
  - 단지, 적당한 곳에 형 매개변수를 포함하는 것이 요구됨
- 템플릿 클래스가 프렌드를 갖는 것은 매우 일반적
  - 특히 연산자 오버로드에 대해 (보았던 것처럼)

사전 정의 템플릿 클래스들

- vector 클래스도 템플릿 클래스!
- 다른 것: basic_string 템플릿 클래스
  - 어떤 형의 원소도 문자열로 취급

basic_string<char>	char에 동작
basic_string<double>	doubles에 동작
basic_string<YourClass>	YourClass 객체에 동작

  - "string " 은 basic_string<char>를 위한 또 다른 이름
  - 모든 멤버 함수들이 basic_string<T>에 대해 유사하게 작동함
  - basic_string은 <string> 라이브러리에 정의됨
    - 정의는 std 네임스페이스 안에

템플릿과 상속

- 일반 클래스로부터 일반 클래스를 파생시키는 것과 같은 방법
- 파생된 템플릿 클래스
  - 템플릿 또는 템플릿이 아닌 클래스로부터 파생될 수 있음
  - 파생된 클래스는 자연적으로 템플릿 클래스임

요약

- 함수 템플릿
  - 형에 대한 매개변수가 있는 함수를 정의
- 클래스 템플릿
  - 클래스의 일부분을 위한 형 매개변수를 가지는 클래스 정의
- 사전에 정의된 vector와 basic_string 클래스는 템플릿 클래스
- 템플릿 기반 클래스로부터 파생되는 템플릿 클래스를 정의 가능