항목 47. 타입에 대한 정보가 필요하다면 특성정보 클래스를 사용하자

STL은 기본적으로 컨테이너(container), 반복자(iterator), 알고리즘(algorithm), 유틸리티(utility)들로 구성되어 있다.

STL 반복자는 지원하는 연산에 따라 다섯 개의 범주로 나뉜다.

1. 입력 반복자(input iterator) : 전진만 가능, 한칸씩 이동, 현재위치 읽기만 가능하고 한번만 가능 ex)istream\_iterator

2. 출력 반복자(output iterator) : 입력 반복자와 비슷하지만 출력 만 가능 ex) ostream\_iterator

3. 순방향 반복자(forward iterator) : 입력 반복자 + 출력 반복자, 현재 위치 읽기 쓰기를 동시에 여러 번 가능. 다중 패스(multi-pass) 알고리즘 문제에 쓸 수 있음 ex) TR1의 해시 컨테이너 반복자

4. 양방향 반복자(bidirectional iterator) : 순방향 반복자 + 뒤로 가는 기능추가 ex) STL의 list, set, multiset, map, multimap

5. 임의 접근 반복자(random access iterator) : 양방향 반복자 + 반복자 산술 연산(iterator arithmetic) 수행 기능 추가 ex) vector, deque, string

Advance 라는 이름의 지정된 반복자를 지정된 거리 만큼 이동시키는 템플릿을 구현할 경우 위 5가지 모두에 대해서 순차적으로 이동하게 구현을 한다면 임의 접근 반복자의 경우 분명 손해를 본다.

이럴 경우 **특성정보(traits)**를 구현하면 해결 방법이 생긴다. 특성정보란 컴파일 도중에 어떤 주어진 타입의 정보를 얻을 수 있게 하는 객체를 지칭 하는 개념이다. C++ 에 미리 정의된 구조가 아니라 C++ 프로그래머들이 따르는 구현기법이다. 여기에는 몇 가지 요구사항이 있다. 그 중 가장 중요한 하나가 기본제공 타입과 사용자 정의 타입에서 모두 돌아가야 한다. 즉 포인터 등의 기본제공 타입에 적용할 수 있어야 한다는 것이다. 그런데 포인터나 int등의 기본 타입은 특성정보를 그 타입내부에 가지고 있을 수 없다. 즉, 특성 정보는 그 타입의 외부에 존재해야 한다. 가장 표준적인 방법은 특성정보를 템플릿 및 그 템플릿의 1개 이상의 특수화 버전으로 구현하는 것이다. 그리고 보통 struct로 구현한다. 그 구조체를 가리켜 ‘특성 정보 클래스’라고 부른다.

STL 반복자는 아래와 같이 구현되어 있다.

|  |
| --- |
| struct input\_iterator\_tag{};  struct output\_iterator\_tag{};  struct forward\_iterator\_tag : public input\_iterator\_tag{};  struct bidirectional\_iterator\_tag: public forward\_iterator\_tag{};  struct random\_access\_iterator\_tag: public bidirectional\_iterator\_tag{};  template<...>  class deque  {  public:  class iterator{  public:  typedef random\_access\_iterator\_tag iterator\_category;  };  };  template<...>  class list{  public:  class iterator{  public:  typedef bidirectional\_iterator\_tag iterator\_category;  };  }; |

클래스 내부 typedef를 앵무새처럼 똑같이 재생한 것이 iterator\_trait 이다. 포인터 타입에 대해서는 **부분 템플릿 특수화(partial template specialization)** 버전을 제공한다.

|  |
| --- |
| template<typename iterT>  struct iterator\_traits {  typedef typename IterT::iterator\_category iterator\_category;  ...  };  template<typename IterT>  struct iterator\_traits<IterT\*>  {  typename random\_access\_iterator\_tag iterator\_category;  ...  } |

특성정보 클래스의 설계 및 구현 방법은 다음과 같이 정리할 수 있다.

**- 다른 사림이 사용하도록 열어 주고 싶은 타입 관련 정보를 확인한다. (반복자 범주)**

**- 그 정보를 식별하기 위한 이름을 선택한다. (iterator\_category)**

**- 지원하고자 하는 타입 관련 정보를 담은 템플릿 및 템플릿의 특수화 버전을 제공**

이제 advance의 의사코드를 다음과 같이 구현하면 될 것이다.

|  |
| --- |
| template<typename IterT, typename DistT>  void advance(IterT& iter, DistT d)  {  if(typeid(typename std::iterator\_traits<IterT>::iterator\_category) == typeid(std::random\_access\_iterator\_tag))  ...  } |

하지만 위와 같이 구현하면 컴파일 도중에 할 수 있는 것을 굳이 실행 도중에 하도록 되기 때문에 진정한 시간 낭비가 아닐 수 없을 뿐더러 실행 코드의 크기도 비대해진다. 이것을 개선할 수 있는 방법은 바로 **오버로딩**이다. 즉 매개변수는 다르지만 이름이 같은 함수를 이용하는 것이다.

|  |
| --- |
| template<typename T, typename D>  void doAdvance(T& t, D d, std::random\_access\_iterator\_tag){  t += d;  }  template<typename T, typename D>  void doAdvance(T& t, D d, std::random\_access\_iterator\_tag){  if (d >= 0) { while (d--) ++t; }  else { while (d++) --t; }  }  template<typename T, typename D>  void doAdvance(T& t, D d, std::input\_iterator\_tag){  if( d < 0 ) { throw std::out\_of\_range(“Negatice distance”); }  while (d--) ++t;  }  template<typename T, typename D>  void doAdvance(T& t, D d){  doAdvance(t, d, typename std::iterator\_traits<T>::iterator\_category);  } |

특성정보 클래스를 어떻게 사용하는지 깔끔하게 정리해 보겠다.

- **작업자(worker)** 역할을 마을 함수 혹은 함수 템플릿 (doAdvance)을 특성정보 매개변수를 다르게 하여 오버로딩하고, 각 특성에 맞추어 구현한다.

- 작업자를 호출하는 **주인(master)** 역할을 맡을 함수 혹은 함수 템플릿(advance)을 만들고 특성정보 클래스에서 제공하는 정보를 넘겨서 작업자를 호출한다.

C++ 표준 라이브러리에 특성정보는 엄청 많다. 그 중 대표적인게 value\_type, char\_traits, numeric\_limits 등이 있다. TR1에는 is\_fundamental<T>, is\_array<T>, is\_base\_of<T1, T2> 등 50여가지가 있다.

**요약.**

- 특성정보 클래스는 컴파일 도중에 사용할 수 있는 타입 관련 정보를 만들어냅니다. 또한 특성정보 클래스는 템플릿 및 템플릿 특수 버전을 사용하여 구현합니다.

- 함수 오버로딩 기법과 결합하여 특성정보 클래스를 사용하면, 컴파일 타임에 결정되는 타입별 if . . . else 점검문을 구사할 수 있습니다.