CHAPTER 17
STL

학습 목표

- STL의 4가지 구성 요소(컨테이너, 알고리즘, 반복자, 함수 객체)를 알아봅니다.
- 반복자의 종류를 알아봅니다.
- 시퀀스 컨테이너인 벡터(vector), 덱(deque), 리스트(list)를 이해합니다.
- 컨테이너 어댑터인 스택(stack), 큐(queue), 우선순위 큐(priority queue)를 이해합니다.
- 연관 컨테이너인 세트(set), 맵(map)을 이해합니다.
- 이번 장에서 소개하는 알고리즘을 활용하기 위해 함수와 함수 객체를 사용하는 방법을 파악합니다.
- STL에 정의되어 있는 일반적인 알고리즘에 대해서 알아봅니다.

01 STL 소개

1 STL의 구성 요소

표준 템플릿 라이브러리STL: Standard Template Library는 소프트웨어 재사용성과 기능 분리라는 2가지 문제를 해결하기 위해 많은 개발자가 수 년에 걸쳐서 만들어낸 연구 결과라고 할 수 있습니다. STL은 [그림 19-1]과 같은 4가지 구성 요소로 이루어집니다.

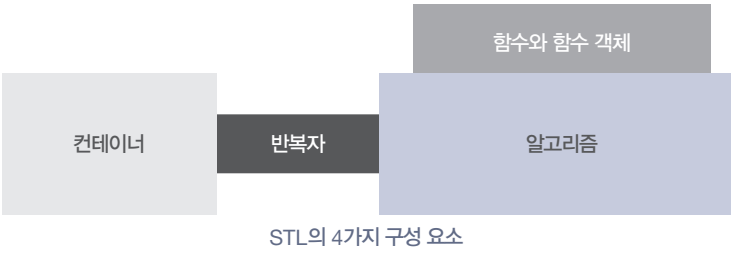


그림 19-1 STL의 구성 요소

컨테이너

컨테이너container는 STL의 핵심입니다. 시퀀스 컨테이너sequence container, 컨테이너 어댑터container adapter, 연관 컨테이너associate container, 의사 컨테이너pseudo container라는 목적에 따라 만들어진 4가지 컨테이너로 구분할 수 있습니다. 이 책에서는 앞에 있는 3가지 컨테이너만 살펴봅니다.

알고리즘

알고리즘algorithm은 컨테이너 요소에 적용할 연산을 의미합니다. 알고리즘은 크게 4가지로 구분할 수 있습니다.

- 비변경 알고리즘non-mutating algorithm은 컨테이너 구조를 변경하지 않습니다.
- 변경 알고리즘mutating algorithm은 컨테이너 구조를 변경합니다.
- 정렬 알고리즘sorting algorithm은 컨테이너의 요소를 정렬합니다.
- 수치 알고리즘numeric algorithm은 숫자 요소에 수학 처리를 합니다.

반복자

컨테이너와 컨테이너 내부의 요소는 서로 자료형에 영향을 주지 않습니다. 예를 들어서 물건을 정리하는 경우를 생각해봅시다. 상자에 책을 넣을 수도 있고, 장난감을 넣을 수도 있고, 돌을 넣을 수도 있고, 캔 음료를 넣을 수도 있습니다. 반대로 책을 상자에 넣을 수도 있고, 가방에 넣을 수도 있고, 책장에 넣을 수도 있습니다. 어떤 곳에 어떤 것을 넣어도 상관없이 무언가를 저장할 수 있고, 하나하나 접근할 수 있습니다.

STL는 반복자iterator라는 것을 사용해 요소 하나하나에 접근해서 처리를 합니다. 반복자는 컨테이너의 자료형 또는 요소의 자료형과 상관없이 모두 사용할 수 있습니다. 컨테이너가 반복자를 지원하기만 한다면 어떤 것을 활용해도 알고리즘을 적용할 수 있습니다.

함수와 함수 객체

STL에는 여러 범용 알고리즘이 있습니다. 그리고 필요한 경우에는 함수 또는 함수 객체를 사용해 사용자 정의 알고리즘을 정의할 수 있습니다. 함수를 활용할 때는 정해진 형태의 함수를 만들어야 하며, 함수 객체를 활용할 때는 operator() 연산자 오버로드가 정의된 클래스를 만들어야 합니다.

이번 장에서 살펴보는 모든 예시는 STL을 사용합니다. 일부 예시는 이전에 살펴보았던 예시를 STL로 구현하는 예제이고, 일부 예시는 이번 절에서 새로 나옵니다.

02 반복자

1 반복자 구조

반복자iterator는 포인터를 추상화한 것입니다. 반복자는 데이터 멤버로 포인터를 가지며, 추가적으로 이를 활용하는 여러 멤버 함수를 갖습니다. 포인터는 C++이 제공하는 문법이므로 추가적인 제한을 걸거나 기능을 추가할 수 없습니다. 하지만 반복자라는 객체로 포인터를 추상화하면, 추가적인 제한을 걸거나 기능을 추가할 수 있습니다.

예를 들어서 반복자에 + 연산자와 − 연산자에 제한을 걸어서 다른 객체를 임의로 선택할 수 없게 만들 수도 있고, ++ 연산자와 −− 연산자를 추가해서 앞뒤의 요소를 선택하게 만들 수도 있습니다. 또한 요소에 접근만 하게 만들 수도 있고, 접근과 함께 변경도 하게 만들 수도 있습니다.

또한 반복자를 사용하면 컨테이너의 내부 구조를 사용자에게 숨기고, 사용만 하게 만들 수 있습니다. [그림 19−2]처럼 대부분의 컨테이너는 내부적으로 조작할 수 없는 내부 반복자를 갖고 있습니다. 그리고 사용자는 외부 반복자라는 것을 사용해서 컨테이너 내부의 요소에 접근합니다. 내부 반복자는 외부 반복자를 처리할 때 사용됩니다.

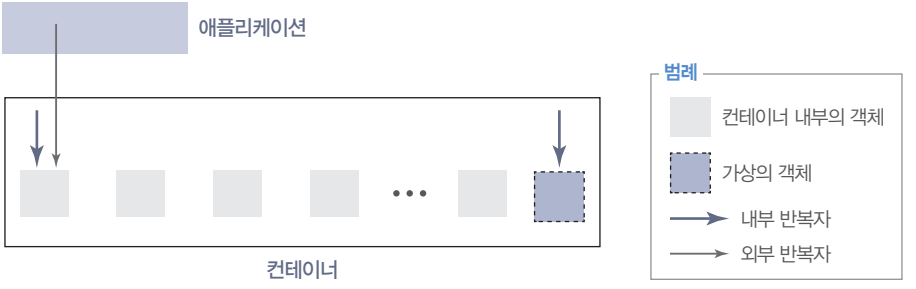


그림 19−2 내부 반복자와 외부 반복자

그림을 보면 왼쪽에 있는 내부 반복자는 컨테이너의 첫 번째 객체를 가리키고, 오른쪽에 있는 내부 반복자는 컨테이너 끝부분의 존재하지 않는 위치를 가리킵니다. 외부 반복자는 처음 초기화될 때 왼쪽 내부 반복자를 가리킵니다. 애플리케이션에서는 이를 오른쪽 내부 반복자까지의 범위 내부에서 옮길 수 있으며, 이를 활용해서 컨테이너 내부의 객체에 접근합니다. 이러한 외부 반복자를 사용할 때는 컨테이너의 내부 구조를 파악하지 않아도 됩니다.

2 반복자 종류

반복자는 [그림 19-3]처럼 입력 반복자, 출력 반복자, 전방 반복자, 양방향 반복자, 임의 접근 반복자로 구분할 수 있습니다.

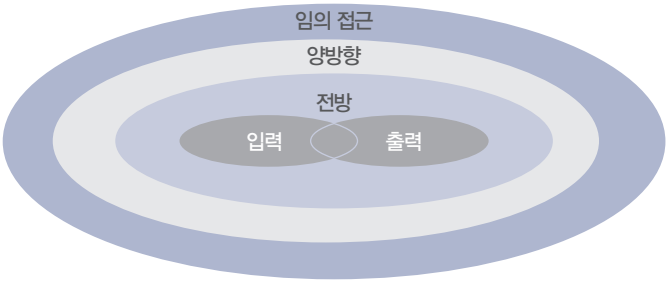


그림 19-3 반복자 종류의 계층

표 19-1 반복자의 종류

반복자	읽기	쓰기	*	++	--	==, !=	<, <=, >, >=	+-
입력	V		V	V		V		
출력		V	V	V		V		
전방	V	V	V	V		V		
양방향	V	V	V	V	V	V		
임의 접근	V	V	V	V	V	V	V	V

입력 반복자

입력 반복자input iterator는 요소를 읽을 때 사용하며 읽기만 가능합니다. 조금 어렵게 표현하면, 입력 반복자는 컨테이너를 데이터를 읽어 들이는 소스로 사용하는 반복자입니다.

출력 반복자

출력 반복자output iterator는 요소를 쓸 때 사용합니다. 쓰기만 가능합니다.

전방 반복자

전방 반복자forward iterator는 요소를 읽고 쓸 때 사용합니다. 기능적으로 입력 반복자와 출력 반복자가 결합된 것입니다.

양방향 반복자

양방향 반복자bidirectional iterator는 양방향으로 움직일 수 있는 반복자입니다. ++ 연산자와 -- 연산자가 정의를 사용해 전방 또는 후방으로 이동할 수 있습니다. 전방과 후방이 어떤 방향을

나타내는지는 이후에 예시를 살펴보면서 알아봅니다.

임의 접근 반복자

임의 접근 반복자random-access iterator는 양방향 접근자에 [ ], +, −, 〈, 〈=, 〉, 〉= 연산자를 추가한 반복자입니다. 임의 접근 반복자는 이러한 연산자를 사용해서 한 번에 원하는 위치의 요소에 접근할 수 있습니다.

3 이동 방향

컨테이너는 기본적으로 기본 반복자iterator와 역 반복자reverse_iterator라는 2가지 종류의 반복자를 정의하고 있습니다. 이러한 반복자의 이동 방향은 [그림 19-4]와 같습니다.

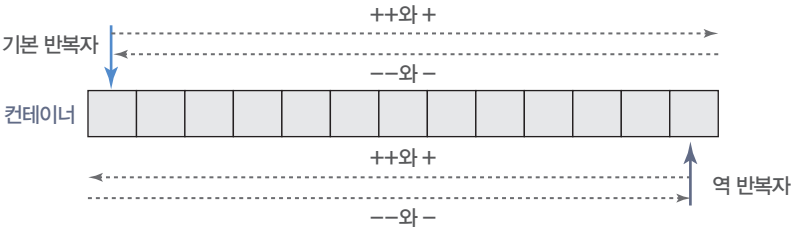


그림 19-4 기본 반복자와 역 반복자의 이동 방향

기본 반복자의 ++와 +는 뒤로 이동, −−와 −는 앞으로 이동을 나타냅니다. 역 반복자의 ++와 +는 앞으로 이동, −−와 −는 뒤로 이동을 나타냅니다. 기본 반복자와 역 반복자는 이렇게 이동 방향이 서로 반대입니다.

4 상수성

데이터를 변경할 수 있는지 없는지를 상수성constantness이라고 표현합니다. 컨테이너에는 이러한 상수성에 따라서 const iterator와 const_iterator가 정의되어 있습니다.

자료형 const iterator

const iterator는 한 번 어떤 요소를 가리키면, 다른 요소를 가리킬 수 없습니다. 마치 배열의 이름처럼 고정된 곳을 가리킵니다.

자료형 const_iterator

const_iterator 자료형은 요소를 rvalue로 역 참조합니다. 따라서 요소를 변경할 수 없습니다. 마치 값을 변경할 수 없는 상수로 배열을 선언한 것과 비슷합니다.

03 시퀀스 컨테이너

1 개요

시퀀스 컨테이너sequence container는 프로그래머가 요소를 저장하고 요소를 찾는 순서를 제어할 수 있는 컬렉션(객체의 모음)입니다. STL은 벡터vector, 덱deque, 리스트list라는 3가지 시퀀스 컨테이너를 제공합니다. 벡터와 덱은 동적 배열로 구현되어 있으며, 리스트는 이중 링크드 리스트로 구현되어 있습니다.

2 공용 인터페이스

[표 19-2]의 공용 인터페이스를 살펴봅시다. SC는 vector, deque, list로 대체해서 생각하세요. 약자 V는 vector, D는 deque, L은 list를 나타냅니다. T 자료형은 내장 자료형과 사용자 정의 자료형 모두 가능합니다. iter는 반복자, inIter은 입력 반복자, pred는 true 또는 false를 나타내는 불 함수입니다.

표 19-2 시퀀스 컨테이너의 공용 인터페이스

생성자, 할당 연산자, 소멸자	V	D	L
SC<T>::SC()	V	V	V
SC<T>::SC(size_type n, const T& value = T())	V	V	V
SC<T>::SC(const_iter first, const_iter last)	V	V	V
SC<T>::SC(const SC <T>& other)	V	V	V
SC<T>& SC<T>::operator =(const SC<T>& other)	V	V	V
SC<T>::~SC()	V	V	V
크기와 용적	**V**	**D**	**L**
size_type SC<T>::size()	V	V	V
size_type SC<T>::max_size()	V	V	V
void SC<T>::resize(size_type n, T value = T())	V	V	V
bool SC<T>::empty()	V	V	V
size_type SC<T>::capacity()	V		

void SC<T>::reserve(size_type, n)	V		
요소 접근(const iterator 형태도 있음)	**V**	**D**	**L**
T& SC<T>::front()	V	V	V
T& SC<T>::back()	V	V	V
T& SC<T>::operator[]()(size_type index)	V	V	
T& SC<T>::at(size_type index)	V	V	
반복자(const_iterator 형태도 있음)	**V**	**D**	**L**
iter SC<T>::begin()	V	V	V
iter SC<T>::end()	V	V	V
reverse_iter SC<T>::rbegin()	V	V	V
reverse_iter SC<T>::rend()	V	V	V
삽입	**V**	**D**	**L**
void SC<T>::push_front(const T& value)		V	V
void SC<T>::push_back(const T& value)	V	V	V
iter SC<T>::insert(iter pos, const T& value)	V	V	V
void SC<T>::insert(iter pos, size_type n, const T& value)	V	V	V
void SC<T>::insert(iter pos, InIter first, inIter last)	V	V	V
삭제	**V**	**D**	**L**
void SC<T>::pop_front()		V	V
void SC<T>::pop_back()	V	V	V
iter SC<T>::erase(iter pos)	V	V	V
iter SC<T>::erase(iter first, iter second)	V	V	V
void SC<T>::clear()	V	V	V
void SC<T>::remove(const T& value)			V
void SC<T>::remove_if(pred p)			V
void SC<T>::unique(pred p)			V
접합, 결합, 정렬	**V**	**D**	**L**
void SC<T>::splice(iter pos, SC<T> other)			V
void SC<T>::splice(iter pos, SC<T> other, iter other)			V
void SC<T>::splice(iter pos, SC<T> other, iter i, iter j)			V
void SC<T>::merge(SC<T> other)			V
void SC<T>::sort()			V

스왑	V	D	L
void SC<T>::swap(SC<T>& other)	V	V	V
전역 함수(op는 <, <=, >, >=, ==, !=)	**V**	**D**	**L**
bool operator op(const SC<T> left, const SC<T> right)	V	V	V
void swap(SC<T>& left, SC<T>& right)	V	V	V

3 vector 클래스

〈vector〉 헤더 파일에 정의되어 있는 vector 클래스(벡터)는 빠른 임의 접근을 할 수 있으며, 컨테이너의 뒤에 빠른 삽입과 삭제를 특징으로 하는 시퀀스 컨테이너입니다. 반면 컨테이너의 앞과 중간에 요소를 추가하거나 제거하는 경우에는 적합하지 않습니다. 벡터를 그림으로 나타내면 [그림 19-5]와 같습니다.

그림 19-5 인덱스를 사용해 접근할 수 있는 시퀀스 컨테이너인 벡터

벡터는 힙 메모리에 할당된 배열을 활용해서 구현되었으며, 다양한 연산을 제공해줍니다. 벡터는 배열처럼 인덱스를 사용해 요소에 접근할 수 있습니다. 반면 배열과 다르게 필요한 경우에 크기를 변경할 수 있습니다.

연산

벡터는 [표 19-2]에 있던 공용 인터페이스의 멤버 함수들을 사용할 수 있습니다. 자주 사용되는 연산 몇 가지를 살펴봅시다.

■ 생성자와 할당 연산자

벡터는 기본 생성자 1개, 매개변수가 있는 생성자 2개, 복사 생성자 1개, 할당 연산자 1개를 갖습니다. 코드는 다음과 같이 사용됩니다.

```
vector<T> vec;            // 빈 벡터를 생성
vector<T> vec(4, value);  // value라는 값을 갖는 4개의 요소를 가진 벡터를 생성
```

```
vector<T> vec(from, to);     // 다른 시퀀스 컨테이너를 기반으로 벡터를 생성
vector<T> vec(otherVec);     // 복사 생성자
vector<T> vec = otherVec;    // 할당 연산자
```

첫 번째 줄은 빈 벡터를 생성합니다. 두 번째 줄은 특정 값을 갖는 요소 4개를 생성합니다. 만약 두 번째 매개변수를 입력하지 않으면, 기본값(정수 벡터라면 0)이 들어갑니다. 세 번째 줄은 다른 시퀀스 컨테이너를 기반으로 벡터를 만들 때 사용합니다. from과 to는 다른 컨테이너에서 복사하고 싶은 위치의 시작과 끝을 가리키는 반복자입니다. 이때 from은 포함하지만 to는 포함하지 않습니다. 네 번째 줄은 다른 벡터를 복사하는 복사 생성자입니다. 그리고 다섯 번째 줄의 할당 연산자는 복사 생성자와 같은 역할을 합니다. 다만 vec 객체가 이미 존재하는 객체여야 합니다.

■ 소멸자

소멸자는 벡터가 스코프를 벗어날 때 자동으로 호출됩니다. 소멸자가 호출되면 힙에 할당된 모든 배열을 해제합니다.

■ 크기와 용적

[표 19-2]의 공용 인터페이스에서 볼 수 있는 것처럼 벡터에는 크기, 용적과 관련된 멤버 함수가 6개 있습니다. 코드로 나타내면 다음과 같습니다. 다음 코드에서 vec은 벡터 클래스를 인스턴스화한 벡터 객체를 의미합니다.

```
vec.size();                  // 현재 크기를 리턴
vec.max_size();              // 최대 크기를 리턴
vec.resize(n, value);        // 벡터의 크기를 변경
vec.empty();                 // 빈 벡터라면 true를 리턴
vec.capacity();              // 용적을 리턴
vec.reserve(n);              // 메모리 위치를 추가 할당
```

size 함수는 현재 벡터의 요소 수를 리턴합니다. max_size 함수는 벡터가 가질 수 있는 요소의 최대 수를 리턴합니다. STL 내부에 정의되어 있는 매우 큰 숫자가 나올 것입니다. empty 함수는 배열의 크기가 0일 때 true를 리턴합니다.

resize(n, value) 함수는 n이 배열의 크기보다 크면(n 〉 size), 배열을 확장하고 추가된 요소를 value 값으로 초기화합니다. n이 배열의 크기보다 작으면(n 〈 size), 요소를 제거합니다. value를 지정하지 않으면 기본값이 사용됩니다. capacity 함수는 용적을 리턴합니다.

reserve 함수는 용적을 크게 예약할 때 사용합니다. 벡터는 기본적으로 메모리에 필요한 만큼의 용적을 할당하고, 필요하면 용적을 늘리거나 줄입니다. 이때 내부적으로 배열을 새로 만들고, 배열을 복사하고, 기존의 배열을 삭제하는 처리를 합니다. 이러한 작업이 너무 많이 반복되면 성능이 좋지 않아집니다. 그래서 필요한 만큼 한 번에 큰 용적을 명시적으로 추가할 때 reserve 함수를 사용합니다.

■ 요소에 접근(추출과 변경)

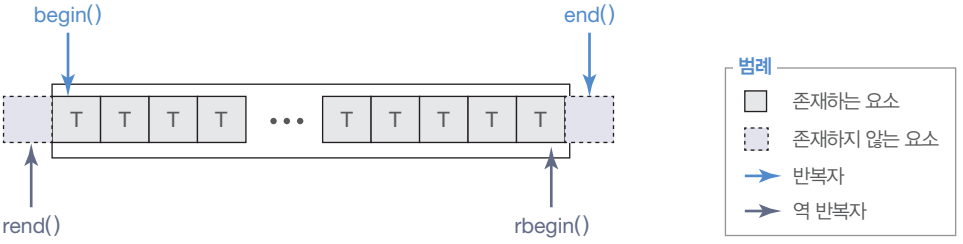
[표 19-2]의 공용 인터페이스에서 볼 수 있는 것처럼 벡터는 요소에 접근할 때 다음과 같은 코드를 사용합니다.

```
vec.front();      // 가장 앞의 요소에 접근
vec.back();       // 가장 마지막의 요소에 접근
vec[i];           // i 번째 요소에 접근
vec.at(i);        // i 번째 요소에 접근
```

vec[i]와 vec.at(i) 모두 i 번째 요소에 접근할 때 사용할 수 있습니다. vec.at(i) 형태가 범위를 확인해서 index of out range 예외를 발생시켜주므로, 일반적으로 vec.at(i) 형태를 사용합니다.

■ 반복자

벡터는 [그림 19-6]처럼 2개의 기본 반복자와 2개의 역 반복자를 제공합니다. [표 19-2]에는 기본 반복자 하나만 표시했습니다.



begin() 함수는 가장 앞에 존재하는 요소를 가리키는 반복자를 리턴
end() 함수는 마지막 요소 뒤의 존재하지 않는 요소를 가리키는 반복자를 리턴
rbegin() 함수는 가장 뒤에 존재하는 요소를 가리키는 반복자를 리턴
rend() 함수는 가장 앞의 존재하지 않는 요소를 가리키는 반복자를 리턴
두 반복자 모두 임의 접근 반복자이므로 앞뒤로 이동 가능

그림 19-6 벡터의 반복자와 역 반복자

begin(), end(), rbegin(), rend()로 리턴되는 반복자는 고정된 반복자이며, 이동시킬 수 없습니다. 이러한 반복자는 바른 반복자의 범위 등을 확인할 때 사용합니다. 컨테이너에 반복자로 반복을 돌리려면, 별도로 반복자를 인스턴스화해서 사용해야 합니다. 클래스별로 기본 반복자와 역 반복자가 제공되므로 필요에 따라서 사용하면 됩니다. 일반적으로 코드에서 기본 반복자는 iter, 역 반복자는 riter로 사용합니다.

```
vector<T>::iterator iter;              // 기본 반복자
vector<T>::reverse_iterator riter;     // 역 반복자
```

사용자 반복자를 인스턴스화했다면, 일단 반복자가 어떤 위치를 가리키게 해야 합니다. iter 반복자는 begin() 함수가 리턴하는 위치로, riter 반복자는 rbegin() 함수가 리턴하는 위치로 할당해서 사용합니다.

```
iter = vec.begin();    // iter는 vec.begin()이 가리키는 위치부터 반복을 시작하게 함
riter = vec.rbegin();  // riter는 vec.rbegin()이 가리키는 위치부터 반복을 시작하게 함
```

[프로그램 19-1]은 크기가 10인 벡터를 만들고, 반복자와 역 반복자를 사용해서 요소에 접근하는 예입니다.

프로그램 19-1 벡터의 반복자 사용하기 Prg19-1.cpp

```
1  /*************************************************************************
2   * 반복자를 사용해보는 간단한 프로그램                                   *
3   *************************************************************************/
4  #include <vector>
5  #include <iostream>
6  #include <iomanip>
7  using namespace std;
8
9  int main()
10 {
11     // 10개의 요소를 가진 벡터와 반복자 2개 생성
12     vector<int> vec(10);
13     vector<int>::iterator iter;
14     vector<int>::reverse_iterator rIter;
15     // 요소의 값을 변경
16     for(int i = 0; i < 10; i++)
```

```
17      {
18          vec.at(i) = i * i;
19      }
20      // 기본 반복자로 요소 출력
21      cout << "기본 탐색: ";
22      for(iter = vec.begin(); iter != vec.end(); ++iter)
23      {
24          cout << setw(4) << *iter;
25      }
26      cout << endl;
27      // 역 반복자로 요소 출력
28      cout << "역 탐색: ";
29      for(rIter = vec.rbegin(); rIter != vec.rend(); ++rIter)
30      {
31          cout << setw(4) << *rIter;
32      }
33      cout << endl;
34      return 0;
35  }
```

실행 결과

```
기본 탐색:    0   1   4   9  16  25  36  49  64  81
역 탐색:   81  64  49  36  25  16   9   4   1   0
```

기본 반복자는 vec.begin()부터 시작하고 vec.end()에서 종료되며, 역 반복자는 vec.rbegin()부터 시작해서 vec.rend()에서 종료됩니다. 기본 반복자와 역 반복자 모두 ++ 연산자를 사용해서 위치를 이동합니다. 기본 반복자는 ++하면 벡터의 뒤로 이동하고, 역 반복자는 ++하면 벡터의 앞으로 이동한다는 것을 주의하세요. 이어서 벡터는 요소에 임의 접근할 수 있다는 것을 확인합니다. 다음과 같은 요소를 갖는 배열을 사용합니다.

```
0   10   20   30   40   50   60   70   80   90
```

[프로그램 19-2]는 임의 접근하는 예제입니다. 일단 기본 반복자를 사용해서 40을 곧바로 출력하고, 반복자를 앞으로 옮겨서 20을 출력합니다. 이어서 역 반복자를 사용해서 50을 출력하고, 반복자를 뒤로 옮겨서 70을 출력합니다. 기본 반복자의 +와 -의 방향에 항상 주의하세요.

프로그램 19-2 반복자를 사용한 양방향 임의 접근 Prg19-2.cpp

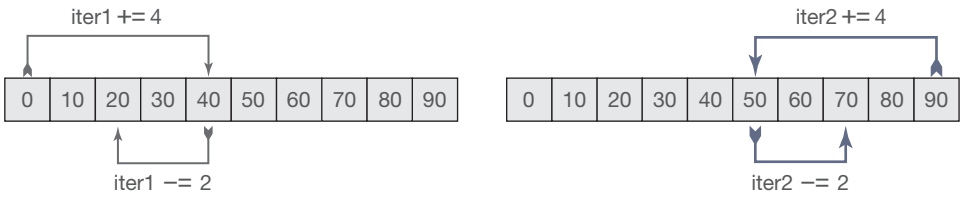
```
/*************************************************************************
 * 임의 접근 기본 반복자와                                               *
 * 역 반복자를 살펴보는 간단한 프로그램                                  *
 *************************************************************************/
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    // 벡터와 반복자 인스턴스화
    vector<int> vec;
    vector<int>::iterator iter1;
    vector<int>::reverse_iterator iter2;
    // 벡터를 10개의 요소로 채우기
    for(int i = 0; i < 10; i++)
    {
        vec.push_back(i * 10);
    }
    // 기본 반복자로 40과 20 출력
    cout << "40 출력 후에 20 출력하기" << endl;
    iter1 = vec.begin();
    iter1 += 4;
    cout << *iter1 << " ";
    iter1 -= 2;
    cout << *iter1 << endl;
    // 역 반복자로 50과 70 출력
    cout << "50 출력 후에 70 출력하기" << endl;
    iter2 = vec.rbegin();
    iter2 += 4;
    cout << *iter2 << " ";
    iter2 -= 2;
    cout << *iter2 << endl;
    return 0;
}
```

실행 결과

```
40 출력 후에 20 출력하기
40  20
50 출력 후에 70 출력하기
50  70
```

[그림 19-7]은 기본 반복자와 역 반복자를 사용해서 임의로 앞뒤로 이동하는 것을 나타낸 그림입니다.



(a) 반복자 사용 (b) 역 반복자 사용

그림 19-7 반복자와 역 반복자의 임의 이동

■ 삽입

벡터 클래스는 하나의 요소 또는 여러 개의 요소를 컨테이너에 추가하는 여러 가지 멤버 함수를 제공합니다. 벡터의 뒤에 요소를 삽입하는 것은 별도의 재할당이 필요 없어서 굉장히 효율적으로 작동합니다. 다만 중간 또는 앞에 요소를 삽입하는 것은 메모리 재할당이 필요하므로 조금 더 느리게 작동합니다. 다음은 멤버 함수를 사용해서 벡터에 새로운 요소를 삽입하는 코드입니다. 마지막 예의 경우 first부터 last 바로 직전까지의 요소를 pos 위치에 삽입합니다.

```
vec.push_back(value);              // 뒤(마지막)에 요소를 추가
vec.insert(pos, value)             // pos 위치에 요소를 추가
vec.insert(pos, n, value);         // pos 위치에 value 값을 갖는 요소 n개를 추가
vec.insert(pos, first, last);      // pos 위치에 [first, last) 위치의 요소를 추가
                                      (last 제외)
```

매개변수 pos는 삽입할 위치를 나타내는 반복자 객체이고, 매개변수 first와 last는 입력 반복자 객체입니다. 네 번째 형태의 경우 다른 컨테이너에 있는 요소를 first 위치부터 last 위치까지 복사해서 추가할 때 사용합니다. last 앞까지만 복사되므로 last 위치의 요소는 포함되지 않습니다.

■ 삭제

벡터는 컨테이너의 요소를 한 개 또는 여러 개 삭제할 수 있는 멤버 함수를 제공합니다. 뒷부분의 요소를 제거하는 일은 배열 재할당이 필요 없으므로 굉장히 효율적입니다. 하지만 앞 또는 중간 부분의 요소를 제거하는 일은 배열 재할당이 동반되므로 피하는 것이 좋습니다.

```
vec.pop_back();              // 뒤(마지막)의 요소를 제거
vec.erase(pos);              // pos 위치의 요소를 제거
vec.erase(first, second);    // first부터 second까지의 요소를 제거
vec.clear();                 // 모든 요소를 제거
```

활용

벡터는 배열처럼 보이게 설계되었습니다. 기본적인 기능은 배열과 같지만, 배열에 비해서 여러 장점을 갖고 있습니다. 첫 번째로 벡터는 여러 연산을 위한 멤버 함수를 갖고 있는 클래스입니다. 두 번째로 힙에 선언되므로 크기를 필요에 따라 변경할 수 있습니다. 세 번째로 굉장히 잘 설계된 반복자 메커니즘을 기반으로 접근, 삽입, 삭제 처리를 할 수 있습니다. 따라서 배열 대신 아무 때나 사용하면 됩니다.

8장에서 2차원 배열을 배웠습니다. 같은 작업을 2차원 벡터로도 만들어서 활용할 수 있습니다. [프로그램 19-3]은 2차원 벡터를 사용해서 곱셈표를 만드는 예제입니다. 코드를 본격적으로 살펴보기 전에 2차원 벡터를 인스턴스화하는 코드를 차근차근 살펴봅시다.

역자 C++11 이전의 컴파일러를 사용하고 있다면, vector<vector<int>>를 작성할 때 오류가 발생합니다. vector<vector<int> >처럼 "> >"의 사이에 띄어쓰기를 넣어야 합니다.

```
vector<type> table(rows, value);
vector<vector<int>> table(rows, value);
vector<vector<int>> table(rows, vector<int>(cols));
```

첫 번째 줄은 type 자료형의 value를 row개 갖는 벡터를 만드는 코드입니다. 두 번째 줄은 type 부분을 vector<int>로 지정했습니다. 세 번째 줄은 value를 vector<int>(cols)로 지정한 것입니다. 따라서 2차원 벡터가 만들어집니다.

프로그램 19-3 벡터의 벡터(테이블) Prg19-3.cpp

```
1  /*************************************************************************
2   * 2차원 벡터로 테이블을 만드는 프로그램                                 *
3   *************************************************************************/
4  #include <vector>
5  #include <iostream>
6  #include <iomanip>
7  using namespace std;
8
9  int main()
```

```
10 {
11     // 벡터의 벡터(2차원 벡터) 생성
12     int rows = 10;
13     int cols = 10;
14     vector<vector<int>> table(rows, vector<int>(cols));
15     // 2차원 벡터의 값 설정
16     for(int i = 0; i < rows; i++)
17     {
18         for(int j = 0; j < cols; j++)
19         {
20             table[i][j] = (i + 1) * (j + 1);
21         }
22     }
23     // 값 추출하여 출력
24     for(int i = 0; i < rows; i++)
25     {
26         for(int j = 0; j < cols; j++)
27         {
28             cout << setw(4) << table[i][j] << " ";
29         }
30         cout << endl;
31     }
32     return 0;
33 }
```

실행 결과

```
   1    2    3    4    5    6    7    8    9   10
   2    4    6    8   10   12   14   16   18   20
   3    6    9   12   15   18   21   24   27   30
   4    8   12   16   20   24   28   32   36   40
   5   10   15   20   25   30   35   40   45   50
   6   12   18   24   30   36   42   48   54   60
   7   14   21   28   35   42   49   56   63   70
   8   16   24   32   40   48   56   64   72   80
   9   18   27   36   45   54   63   72   81   90
  10   20   30   40   50   60   70   80   90  100
```

9장에서 래기드 배열을 사용해 다항식 $(x + y)^n$의 계수를 만들었습니다. [프로그램 19-4]는 래기드 벡터를 사용해서 문제를 해결하는 예시입니다.

프로그램 19-4 래기드 벡터를 사용해서 파스칼의 삼각형 만들기 Prg19-4.cpp

```
/*************************************************************
 * 파스칼의 삼각형을 만드는 프로그램                          *
 *************************************************************/
#include <vector>
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    // 선언
    int power = 5;
    vector<vector<int>> pascal(power + 1, vector<int>());
    // 래기드 벡터 생성
    for(int i = 0; i <= power; i++)
    {
        for(int j = 0; j < i + 1; j++)
        {
            pascal[i].push_back(0);
        }
    }
    // 래기드 벡터를 파스칼의 삼각형으로 채우기
    for(int i = 0; i <= power; i++)
    {
        for(int j = 0; j < i + 1; j++)
        {
            if(j == 0 || i == j)
            {
                pascal[i][j] = 1;
            }
            else
            {
                pascal[i][j] = pascal[i-1][j-1] + pascal[i-1][j];
            }
        }
    }
    // 출력
    for(int i = 0; i <= power; i++)
    {
        cout<< "(x + y)^" << i << "의 계수  =====> ";
        for(int j = 0; j < i + 1; j++)
```

```
42          {
43              cout << setw(4) << pascal[i][j] << " ";
44          }
45          cout << endl;
46      }
47      return 0;
48  }
```

실행 결과

```
(x + y)^0계수  =====>    1
(x + y)^1계수  =====>    1    1
(x + y)^2계수  =====>    1    2    1
(x + y)^3계수  =====>    1    3    3    1
(x + y)^4계수  =====>    1    4    6    4    1
(x + y)^5계수  =====>    1    5   10   10    5    1
```

처음 래기드 벡터를 생성할 때는 생성자에서 (power + 1) 크기의 행을 만들었습니다. 이어서 중첩 반복문으로 push_back 함수를 호출해서 필요한 크기만큼을 0으로 초기화하며 추가했습니다. 이렇게 0으로 초기화한 값은 이어지는 반복문을 돌며 파스칼의 삼각형으로 채워집니다.

4 deque 클래스

〈deque〉 헤더 파일에 정의되어 있는 deque 클래스(덱)는 벡터와 비슷하지만 양쪽으로 요소를 추가할 수 있는 시퀀스 컨테이너입니다. deque라는 이름은 double-ended queue의 약자입니다. 즉 앞과 뒤로 요소를 추가하고 제거할 수 있다는 의미입니다. [그림 19-8]은 n개의 요소를 갖는 덱을 나타낸 그림입니다.

그림 19-8 시퀀스 컨테이너 덱

덱은 앞과 뒤에 요소를 빠르게 추가하고 빠르게 제거할 수 있습니다. 물론 장점이 생기면 단점도 발생합니다. 벡터와 다르게 앞으로도 추가를 할 수 있게 앞쪽으로도 여유 메모리를 할당하기 때문에 메모리를 많이 차지합니다. 따라서 앞의 요소를 추가하거나 제거할 필요가 없다면

벡터를 사용하는 것이 더 효율적입니다. 중간에 요소를 추가하고 제거하는 것은 벡터와 속도 차이가 없습니다.

연산

덱의 멤버 함수는 [표 19-2]의 공용 인터페이스에서 확인할 수 있습니다. 덱의 멤버 함수는 벡터와 유사합니다. 따라서 차이점만 살펴보도록 합시다.

■ 덱에 추가로 있는 연산

덱은 벡터와 비교해서 2개의 멤버 함수가 더 있습니다.

```
deq.push_front(value);          // 앞에 요소를 추가
deq.pop_front();                // 앞에 요소를 제거
```

[표 19-2]의 공용 인터페이스에서 볼 수 있는 것처럼 덱은 capacity 함수와 reserve 멤버 함수가 없습니다. 이는 구현 때문입니다. 덱은 앞뒤로 요소를 추가하고 제거할 수 있으므로, 표준적으로 힙 메모리에 블록 단위로 메모리를 할당하게 구현되어 있기 때문입니다.

활용

덱은 양쪽 끝에 삽입과 삭제를 하는 모든 애플리케이션에서 활용할 수 있습니다. 일반적으로 덱은 목록 내부의 요소를 회전할 때 활용합니다. 시계 방향으로 회전한다면 뒤의 요소를 제거해서 앞에 추가하고, 시계 반대 방향으로 회전한다면 앞의 요소를 제거해서 뒤에 추가합니다. 참고로 덱은 이후에 설명하는 queue 클래스의 베이스 클래스가 됩니다. [프로그램 19-5]는 목록 내부의 요소를 오른쪽으로 밀거나 왼쪽으로 밀어서 회전하게 만드는 예시입니다.

프로그램 19-5 덱으로 요소의 순서 회전하기 Prg19-5.cpp

```
1  /*************************************************************************
2   * 덱으로 요소 회전하는 프로그램                                         *
3   *************************************************************************/
4  #include <deque>
5  #include <string>
6  #include <iostream>
7  #include <iomanip>
8  using namespace std;
9
10 // 전역 print 함수
11 void print(deque<string> deq)
```

```
12 {
13     for(int i = 0; i < deq.size(); i++)
14     {
15         cout << deq.at(i) << "  ";
16     }
17     cout << endl;
18 }
19
20 int main()
21 {
22     // 덱을 생성하고 요소 5개로 초기화
23     deque<string> deq(7);
24     string arr[5] = {"John", "Mary", "Rich", "Mark", "Tara"};
25     for(int i = 0; i < 5; i++)
26     {
27         deq[i] = arr[i];
28     }
29     print(deq);
30     // 시계 방향으로 회전
31     deq.push_back(deq.front());
32     deq.pop_front();
33     print(deq);
34     // 시계 반대 방향으로 회전
35     deq.push_front(deq.back());
36     deq.pop_back();
37     print(deq);
38     return 0;
39 }
```

실행 결과

```
John  Mary  Rich  Mark  Tara
Mary  Rich  Mark  Tara  John
John  Mary  Rich  Mark  Tara
```

23~29행에서 덱에 문자열 5개를 만들었습니다. 31~33행에서 앞의 요소를 복사하고, 뒤에 푸시한 뒤 제거합니다. 35~57행에서는 뒤의 요소를 복사하고 앞에 삽입(푸시)하고 제거(팝)합니다.

5 list 클래스

〈list〉 헤더 파일에 정의되어 있는 list 클래스(리스트)는 빠른 삽입과 삭제가 가능한 시퀀스 컨테이너입니다. 리스트는 목록의 원하는 위치에 요소를 삽입하고, 삭제할 수 있습니다. 다만 리

스트는 이중 링크드 리스트doubly linked list로 구현되어 있어서, 인덱스 또는 at 멤버 함수로 특정 요소에 바로 접근할 수 없습니다. 리스트의 원하는 요소에 접근하려면, 반복자를 사용해서 원하는 요소까지 이동한 뒤 접근해야 합니다. [그림 19-9]는 5개의 요소를 가진 리스트를 나타낸 그림입니다. 노드에는 데이터 부분과 2개의 포인터 부분이 있습니다. 포인터 하나는 이전 노드, 다른 하나는 다음 노드를 가리킵니다.

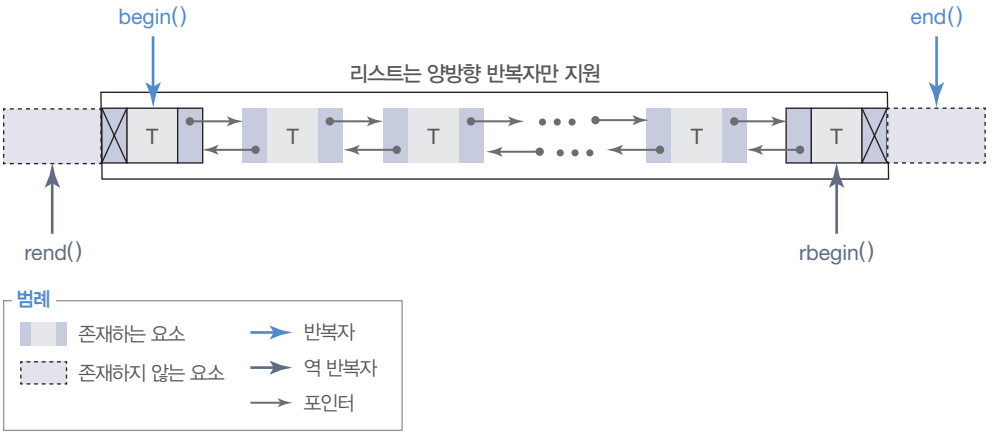


그림 19-9 리스트

연산

리스트의 멤버 함수는 [표 19-2]의 공용 인터페이스에서 확인할 수 있습니다. 리스트의 멤버 함수는 벡터나 덱과 유사합니다. 다만 일부 차이점이 있으므로 차이점만 살펴보도록 합시다.

■ 반복자

리스트는 양방향 반복자만 제공합니다. 임의 접근 반복자가 없으므로, 임의 접근 반복자에 정의되어 있는 + 연산자와 - 연산자를 사용할 수 없습니다. 또한 [ ] 연산자와 at 멤버 함수도 사용할 수 없습니다.

■ 용적과 역순서

리스트는 이중 링크드 리스트를 활용합니다. 따라서 용적이 따로 없고, 원하는 위치에 요소를 추가할 수 있으므로 capacity 함수와 reserve 함수가 없습니다.

■ 요소에 접근

리스트는 임의 접근 연산자를 지원하지 않습니다. 따라서 [ ] 연산자와 at 멤버 함수 등을 사용할 수 없습니다. 요소에 접근하려면 반복자를 명시적으로 사용해야 합니다.

■ **제거**

리스트에는 리스트에서 요소를 지우는 3가지 멤버 함수가 있습니다.

```
remove(value);                  // 모든 value 제거
remove_if(predicate);           // 매개변수가 true를 리턴하는 경우 제거
unique(predicate);              // 중복되는 경우 제거
```

■ **추가 연산**

리스트에는 splice, merge, sort 멤버 함수가 정의되어 있습니다.

```
splice(pos, first, last);       // [first, last)의 요소를 pos 위치에 접합
merge(other);                   // 두 리스트를 합치고 정렬한 새로운 리스트 생성
sort();                         // 리스트 정렬
```

[그림 19-10]은 접합splice의 개념을 나타낸 것입니다. 반복자 pos는 첫 번째 리스트에서 삽입 위치를 지정합니다. first과 last는 두 번째 리스트에서 첫 번째 리스트로 옮길 범위를 지정합니다.

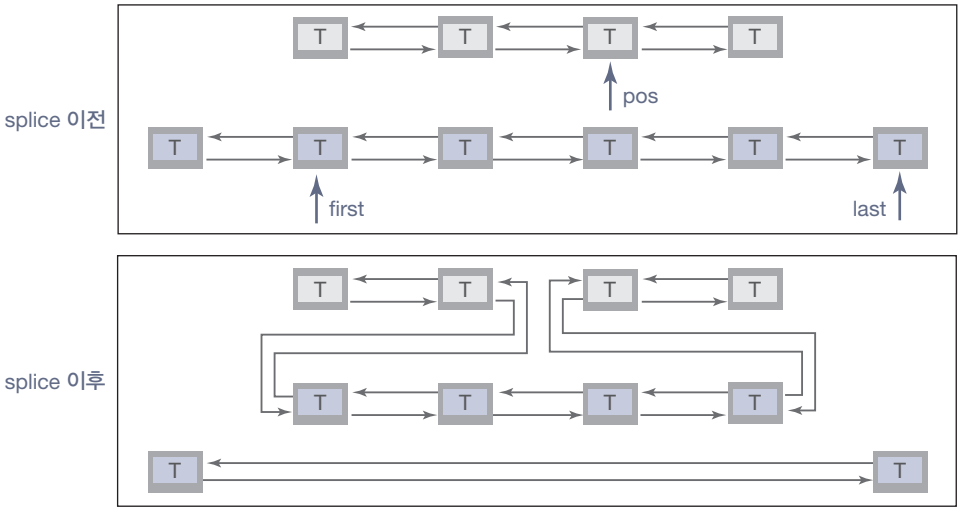


그림 19-10 접합의 개념

활용

list 클래스는 요소를 원하는 위치에 쉽게 추가할 수 있어서, 많은 애플리케이션에서 사용됩니다. 간단한 예를 살펴봅시다.

프로그램 19-6 리스트의 요소 출력하기 Prg19-6.cpp

```
/*************************************************************************
 * 5개의 요소를 갖는 배열을 만들고                                       *
 * 기본 반복자와 역 반복자로 출력하기                                    *
 *************************************************************************/
#include <list>
#include <iostream>
using namespace std;

int main()
{
    // 인스턴스화하고 변수 생성
    list<int> lst;
    int value;
    // 5개의 정수를 입력받고 저장
    for(int i = 0; i < 5; i++)
    {
        cout << "정수를 입력하세요: ";
        cin >> value;
        lst.push_back(value);
    }
    // 기본 방향으로 출력
    cout << "기본 방향" << endl;
    list<int>::iterator iter1;
    for(iter1 = lst.begin(); iter1 != lst.end(); iter1++)
    {
        cout << *iter1 << "  ";
    }
    cout << endl;
    // 역 방향으로 출력
    cout << "역 방향" << endl;
    list<int>::reverse_iterator iter2;
    for(iter2 = lst.rbegin(); iter2 != lst.rend(); iter2++)
    {
        cout << *iter2 << "  ";
    }
    return 0;
}
```

실행 결과

```
정수를 입력하세요: 25
정수를 입력하세요: 32
정수를 입력하세요: 41
정수를 입력하세요: 72
정수를 입력하세요: 95
기본 방향
25  32  41  72  95
역 방향
95  72  41  32  25
```

양방향 반복자가 정의되지 않은 + 연산자와 – 연산자를 사용하지 않고, [프로그램 19-2]를 다시 작성하면 [프로그램 19-7]과 같습니다.

프로그램 19-7 리스트의 요소 일부 출력하기 Prg19-7.cpp

```
1  /*************************************************************************
2   * [프로그램 19-2]를 리스트로 작성한 형태                                *
3   *************************************************************************/
4  #include <iostream>
5  #include <list>
6  using namespace std;
7
8  int main()
9  {
10     // 리스트와 반복자 생성
11     list<int> lst;
12     list<int>::iterator iter1;
13     list<int>::reverse_iterator iter2;
14     // 리스트에 요소 10개 입력
15     for(int i = 0; i < 10; i++)
16     {
17         lst.push_back(i * 10);
18     }
19     // iter1으로 출력
20     cout << "40 출력 후에 20 출력하기" << endl;
21     iter1 = lst.begin();
22     iter1++;
23     iter1++;
24     iter1++;
```

```
25      iter1++;
26      cout << *iter1 << " ";
27      iter1--;
28      iter1--;
29      cout << *iter1 << endl;
30      // iter2로 출력
31      cout << "50 출력 후에 70 출력하기" << endl;
32      iter2 = lst.rbegin();
33      iter2++;
34      iter2++;
35      iter2++;
36      iter2++;
37      cout << *iter2 << " ";
38      iter2--;
39      iter2--;
40      cout << *iter2 << endl;
41      return 0;
42  }
```

실행 결과

```
40 출력 후에 20 출력하기
40   20
50 출력 후에 70 출력하기
50   70
```

자리수가 큰 정수를 처리할 수 있는 BigInteger 클래스를 만듭니다. 두 정수를 더하는 연산만 구현합니다. [프로그램 19-8]은 더하기 연산자(+)만 오버로드하는 BigInteger 클래스입니다. 데이터 멤버로는 자리 수를 저장하는 리스트를 사용합니다. 조금 더 정교하게 만든다면 리스트의 요소에 3자리 또는 6자리의 정수 단위로 저장하게 만들 수 있겠지만 간단하게 요소 하나에 자리 수 하나를 저장하도록 구현합니다.

프로그램 19-8 인터페이스 파일(biginteger.h) Prg19-8.cpp

```
1   /*****************************************************************************
2    * BigInteger 클래스의 인터페이스 파일                                       *
3    *****************************************************************************/
4   #include <string>
5   #include <list>
6   #include <iostream>
7   #ifndef BIGINTEGER_H
```

```
8  #define BIGINTEGER_H
9  using namespace std;
10
11 class BigInteger
12 {
13     private:
14         list<int> lst;
15     public:
16         BigInteger();
17         BigInteger(string str);
18         ~BigInteger();
19         string toString();
20         friend BigInteger operator+(BigInteger first, BigInteger second);
21 };
22 #endif
```

[프로그램 19-9]는 BigInteger 클래스의 구현 파일입니다. 2개의 생성자, 1개의 소멸자, 1개의 덧셈 연산자, 1개의 toString 멤버 함수를 정의합니다. 컴퓨터 프로그래밍에서는 일반적으로 어떤 객체를 문자열로 변환하는 멤버 함수의 이름을 toString이라고 짓습니다.

프로그램 19-9 구현 파일(bigInteger.cpp) Prg19-9.cpp

```
1  /*****************************************************************************
2   * BigInterger 클래스의 구현 파일                                            *
3   *****************************************************************************/
4  #include <iostream>
5  #include <string>
6  #include <iomanip>
7  #include "bigInteger.h"
8  using namespace std;
9
10 // 기본 생성자
11 BigInteger::BigInteger()
12 :lst(list<int>())
13 {
14 }
15 // 매개변수가 있는 생성자
16 BigInteger::BigInteger(string str)
17 :lst(list<int>())
18 {
19     for(int i = 0; i < str.length(); i++)
```

```
20      {
21          int num = str[i] - 48;
22          lst.push_back(num);
23      }
24  }
25  // 소멸자
26  BigInteger::~BigInteger()
27  {
28  }
29  // 리스트를 문자열로 변환하는 함수
30  string BigInteger::toString()
31  {
32      string strg;
33      list<int>::iterator iter;
34      iter = lst.begin();
35      while(iter != lst.end())
36      {
37          strg.append(1, *iter + 48);
38          iter++;
39      }
40      return strg;
41  }
42  // operator+ friend 함수
43  BigInteger operator+(BigInteger first, BigInteger second)
44  {
45      list<int>::reverse_iterator iter1;
46      list<int>::reverse_iterator iter2;
47      BigInteger result;
48      int num1, num2, sum;
49      int carry = 0;
50      iter1 = first.lst.rbegin();
51      iter2 = second.lst.rbegin();
52      while((iter1 != first.lst.rend()) && (iter2 != second.lst.rend()))
53      {
54          num1 = *iter1;
55          num2 = *iter2;
56          sum = (num1 + num2 + carry) % 10;
57          carry = (num1 + num2 + carry) / 10;
58          result.lst.push_front(sum);
59          iter1++;
60          iter2++;
```

```
61      }
62      while((iter1 != first.lst.rend()))
63      {
64          num1 = *iter1;
65          sum = (num1 + carry) % 10;
66          carry = (num1 + carry) / 10;
67          result.lst.push_front(sum);
68          iter1++;
69      }
70      while((iter2 != second.lst.rend()))
71      {
72          num2 = *iter2;
73          sum = (num2 + carry) % 10;
74          carry = (num2 + carry) / 10;
75          result.lst.push_front(sum);
76          iter2++;
77      }
78      if(carry == 1)
79      {
80          result.lst.push_front(carry);
81      }
82      return result;
83  }
```

[프로그램 19-9]의 기본 생성자는 빈 리스트를 만듭니다. 매개변수가 있는 생성자는 '457234 9876509'처럼 큰 정수big integer를 나타내는 문자열을 매개변수로 받고, 여기에서 숫자를 추출합니다. 숫자를 추출할 때는 문자(ASCII 값)에서 48을 빼서 실제 숫자를 나타내는 값으로 변경한 뒤, 이를 리스트에 추가하는 과정을 반복합니다. 생성자를 완료하면 BigInteger 객체는 큰 정수를 나타내는 리스트를 하나 갖습니다. 이어서 toString 멤버 함수는 목록의 숫자를 기반으로 문자열을 조합해서 리턴합니다. 생성자에서 했던 것을 반대로 합니다.

BigInteger 구현의 핵심은 operator+ 연산자 오버로드입니다. operator+ 연산자는 3개의 while 반복문을 사용하고 있습니다. 첫 번째 반복문은 두 객체가 가진 리스트의 길이가 같은 곳까지 실행됩니다. 첫 번째 반복문을 벗어나면 나머지 두 반복문 중 하나가 실행됩니다. 반복문 종료 후에 앞에 자리 수를 추가해야 하는 경우(carry가 있는 경우), 조건문으로 이를 처리하게 했습니다. [프로그램 19-10]은 BigInteger 클래스를 사용하는 애플리케이션 파일입니다.

프로그램 19-10 애플리케이션 파일(app.cpp) Prg19-10.cpp

```
/*****************************************************************************
 * BigInteger 클래스를 사용하는 애플리케이션 파일                              *
 *****************************************************************************/
#include <iostream>
#include <iomanip>
#include "bigInteger.h"
using namespace std;

int main()
{
    // 문자열 입력받기
    string strg1, strg2;
    cout << "첫 번째 큰 정수를 입력하세요: ";
    cin >> strg1;
    cout << "두 번째 큰 정수를 입력하세요: ";
    cin >> strg2;
    // BigInteger 객체 2개 생성
    BigInteger first(strg1);
    BigInteger second(strg2);
    // 두 정수를 더하고 result에 할당
    BigInteger result = first + second;
    // BigInteger 객체를 문자열로 변경
    strg1 = first.toString();
    strg2 = second.toString();
    string strg3 = result.toString();
    string dashes(strg3.length(), '-');
    // 결과 출력
    cout << "첫 번째 정수: " << setw(strg3.length());
    cout << right << strg1 << " + " << endl;
    cout << "두 번째 정수: " << setw(strg3.length());
    cout << right << strg2 << endl;
    cout << "             " << dashes << endl;
    cout << "결과:         " << setw(strg3.length());
    cout << right << strg3 << endl;
    return 0;
}
```

실행 결과

```
첫 번째 큰 정수를 입력하세요: 346786543098762378
두 번째 큰 정수를 입력하세요: 78654329876
첫 번째 정수: 346786543098762378 +
두 번째 정수:         78654329876
             --------------------
결과:         346786621753092254
```

실행 결과

```
첫 번째 큰 정수를 입력하세요: 3478654212345690
두 번째 큰 정수를 입력하세요: 7654329876534567
첫 번째 정수:  3478654212345690 +
두 번째 정수:  7654329876534567
               -----------------
결과:         11132984088880257
```

실행 결과

```
첫 번째 큰 정수를 입력하세요: 356709876567
두 번째 큰 정수를 입력하세요: 19283848484848987654356
첫 번째 정수:             356709876567 +
두 번째 정수: 19283848484848987654356
              ------------------------
결과:         1928384848485205697530923
```

첫 번째 실행 결과에서는 첫 번째 숫자가 더 깁니다. 두 번째 실행에서 자리 수가 올라갔으므로, 앞에서 언급한 carry 부분 조건문이 실행됩니다. 세 번째 실행에서는 첫 번째 정수가 더 짧습니다.

04 컨테이너 어댑터

1 공용 인터페이스

STL은 이전에 살펴보았던 시퀀스 컨테이너보다 가벼운 인터페이스를 가지며 사용하기 쉬운 stack, queue, priority_queue라는 3개의 컨테이너 어댑터container adapter를 제공합니다. 컨테이너 어댑터는 begin, end처럼 반복자를 만드는 멤버 함수를 제공하지 않으므로 알고리즘을 적용할 수 없습니다.

[표 19-3]은 컨테이너 어댑터의 공용 인터페이스입니다. Ad를 stack, queue, priority_queue로 대체해서 생각하세요. 오른쪽에 있는 약자의 S는 stack, Q는 queue, P는 priority_queue를 나타냅니다. T 자료형은 내장 자료형과 사용자 정의 자료형으로 모두 사용 가능합니다.

stack과 priority_queue는 사용하는 상황이 다르지만, 굉장히 비슷한 인터페이스를 갖고 있습니다. stack과 priority_queue의 요소에 접근할 때는 top 멤버 함수만 사용할 수 있습니다. 반면 queue의 요소에 접근할 때는 front와 back을 사용할 수 있습니다.

표 19-3 컨테이너 어댑터 클래스의 공용 인터페이스

생성자	S	Q	P
Ad<T>::Ad()	V	V	V
크기와 용적	**S**	**Q**	**P**
size_type Ad<T>::size() const	V	V	V
bool Ad<T>::empty() const	V	V	V
요소 접근	**S**	**Q**	**P**
T& Ad<T>::front()		V	
T& Ad<T>::back()		V	
T& Ad<T>::top()	V		V
삽입	**S**	**Q**	**P**
void Ad<T>::push(const T& elem)	V	V	V
삭제	**S**	**Q**	**P**
void Ad<T>::pop()	V	V	V

2 stack 클래스

〈stack〉 헤더 파일에 정의되어 있는 stack 클래스(스택)는 삽입push, 삭제pop, 탑top이라는 간단한 기능을 가진 컨테이너 어댑터 클래스입니다. 스택은 한 쪽 끝(탑)으로만 요소를 삽입하고 제거할 수 있습니다. 이처럼 마지막에 추가한 요소가 가장 먼저 제거되는 것을 후입선출LIFO: last-in-first-out이라고 부릅니다.

연산

스택의 생성자는 빈 스택을 생성합니다. 스택은 크기를 확인할 수 있는 size와 empty 멤버 함수를 갖고 있습니다. 스택의 값에 접근할 때 사용할 때는 top 멤버 함수만 사용할 수 있습니다. 값을 넣고 제거할 때는 삽입push과 삭제pop 멤버 함수를 사용합니다.

활용

스택은 후입선출을 활용할 때 사용합니다. 10장에서 사용자 정의 스택 클래스를 활용해서 10진수를 16진수로 변환하는 예제를 살펴보았습니다. 이번에는 [프로그램 19-11]에서 STL을 활용합니다. 18장의 코드와 유사하지만, STL 라이브러리에 있는 스택을 활용하므로 별도의 스택 클래스 정의가 필요 없습니다.

프로그램 19-11 10진수를 16진수로 변환하기 Prg19-11.cpp

```
1  /*************************************************************************
2   * 스택으로 10진수를 16진수로 변환하기                                   *
3   *************************************************************************/
4  #include <stack>
5  #include <iostream>
6  using namespace std;
7
8  int main()
9  {
10     // 스택 인스턴스화
11     stack<char> stk;
12     // 두 문자열과 변수 선언
13     string converter("0123456789ABCDEF");
14     string hexadecimal;
15     int decimal;
16     // 10진수 입력받기
17     do
18     {
19         cout << "양의 정수를 입력하세요: ";
```

```
20          cin >> decimal;
21      } while(decimal <= 0);
22      // 16진수 문자로 변환하고 스택에 입력
23      while(decimal != 0)
24      {
25          stk.push(converter[decimal % 16]);
26          decimal = decimal / 16;
27      }
28      // 스택에서 요소를 꺼내서 문자열에 붙임
29      while(!stk.empty())
30      {
31          hexadecimal.push_back(stk.top());
32          stk.pop();
33      }
34      cout << "16진수 변환 결과 = " << hexadecimal;
35      return 0;
36  }
```

실행 결과

```
양의 정수를 입력하세요: 182
16진수 변환 결과 = B6
```

실행 결과

```
양의 정수를 입력하세요: 1234
16진수 변환 결과 = 4D2
```

실행 결과

```
양의 정수를 입력하세요: 23
16진수 변환 결과 = 17
```

컴파일러를 설계할 때도 스택을 활용합니다. 컴파일러는 표현식에 사용된 괄호가 짝이 맞는지를 확인할 수 있어야 합니다. 다음은 짝이 맞는 괄호와 짝이 맞지 않는 괄호의 예시입니다.

```
(2 + 5) * (3 - 4)
4 + 5 * (6 + 7)
```

■ 짝이 맞는 괄호

```
10 - (5 * (6 + 7)
8 - (5 * 6 + 7) + 4)
```

■ 짝이 맞지 않는 괄호

참고로 이는 여는 괄호와 닫는 괄호의 수가 동일해야 한다는 의미가 아닙니다. 다음과 같이 여는 괄호와 닫는 괄호의 수가 같아도 짝이 안 맞을 수 있습니다.

```
(3 + 4( 7 + 4 ))) 7(8
```

[프로그램 19-12]는 스택을 활용해서 이러한 괄호 짝을 확인하는 프로그램입니다.

프로그램 19-12 괄호 짝 확인하기 Prg19-12.cpp

```
/*************************************************************************
 * 표현식의 괄호 짝이 맞는지 확인하는 프로그램                           *
 *************************************************************************/
#include <stack>
#include <string>
#include <iostream>
using namespace std;

int main()
{
    // 스택, 문자열, 불 선언
    stack<char> stk;
    string expr;
    bool paired = true;
    // 입력받고 스택에 넣고 빼기
    cout << "표현식을 입력하세요:  ";
    getline(cin, expr);
    int i = 0;
    while(i < expr.size() && paired)
    {
        char next = expr[i];
        if(next == '(')
        {
            stk.push(next);
        }
        else if(next == ')')
        {
            if(stk.empty())
            {
                paired = false;  // 스택이 여기에서 비어 버리면 짝이 안 맞음
            }
            else
            {
                stk.pop();
            }
        }
        i++;
    }
```

```
39      // 스택이 비어 있지 않다면 짝이 안 맞는다는 의미
40      if(!stk.empty())
41      {
42          paired = false;
43      }
44      // 결과 출력
45      if(paired)
46      {
47          cout << "괄호 짝이 맞는 정상적인 표현식입니다." << endl;
48      }
49      else
50      {
51          cout << "괄호 짝이 맞지 않습니다!" << endl;
52      }
53      return 0;
54  }
```

실행 결과

```
표현식을 입력하세요: 3 + (4 + 7 + 6)
괄호 짝이 맞는 정상적인 표현식입니다.
```

실행 결과

```
표현식을 입력하세요: 2 + (4 + (6 + 7)
괄호 짝이 맞지 않습니다!
```

실행 결과

```
표현식을 입력하세요:  6 + 7 + (10 * 2 - 4) + 8)
괄호 짝이 맞지 않습니다!
```

3 queue 클래스

〈queue〉 헤더 파일에 정의되어 있는 queue 클래스(큐)는 컨테이너 어댑터 클래스입니다. 큐는 한 쪽 끝으로 요소를 삽입하고, 다른 쪽 끝으로만 제거할 수 있습니다. 이처럼 처음 삽입(푸시)한 요소가 가장 먼저 삭제(팝)되는 것을 선입선출FIFO: first-in-first-out이라고 부릅니다.

연산

큐의 생성자는 빈 큐를 생성합니다. 큐는 크기를 확인할 수 있는 size와 empty 멤버 함수를 갖고 있습니다. 큐의 값에 접근할 때와 사용할 때는 front와 back 멤버 함수를 사용합니다. 값을 넣고 제거할 때는 push와 pop 멤버 함수를 사용합니다.

활용

큐는 선입선출이라는 특성을 활용하여 일반적으로 대기열을 만들 때 활용합니다. 이와 관련된 내용은 18장에서 설명했으므로 생략합니다. 18장에서 큐를 활용해서 만들었던 기부 금액 프로그램을 STL로 만듭니다. 18장의 코드와 유사하지만, STL 라이브러리에 있는 큐를 활용하므로 별도의 큐 클래스 정의가 필요 없습니다.

프로그램 19-13 기부 금액 분류하기 Prg19-13.cpp

```
1  /*************************************************************************
2   * 큐로 기부 내역을 구분하는 프로그램                                    *
3   *************************************************************************/
4  #include <queue>
5  #include <cstdlib>
6  #include <ctime>
7  #include <iostream>
8  using namespace std;
9
10 // print 함수 선언
11 void print(queue<int> queue);
12
13 int main()
14 {
15     // 5개의 큐와 2개의 변수 선언
16     queue<int> queue1, queue2, queue3, queue4, queue5;
17     int num;
18     int donation;
19     // 기부 내역 랜덤하게 생성하면서 구분
20     srand(time(0));
21     for(int i = 0; i < 50; i++)
22     {
23         num = rand();
24         donation = num % (50 - 0 + 0) + 0;
25         switch(donation / 10)
26         {
27             case 0:  queue1.push(donation);
28                      break;
29             case 1:  queue2.push(donation);
30                      break;
31             case 2:  queue3.push(donation);
32                      break;
33             case 3:  queue4.push(donation);
```

```
34                      break;
35              case 4:  queue5.push(donation);
36                      break;
37          }
38      }
39      // 구분된 결과 출력
40      cout << "기부 금액 범위 $00~09: ";
41      print(queue1);
42      cout << "기부 금액 범위 $10~19: ";
43      print(queue2);
44      cout << "기부 금액 범위 $20~29: ";
45      print(queue3);
46      cout << "기부 금액 범위 $30~39: ";
47      print(queue4);
48      cout << "기부 금액 범위 $40~49: ";
49      print(queue5);
50      return 0;
51  }
52  // print 함수의 정의
53  void print(queue<int> queue)
54  {
55      while(!queue.empty())
56      {
57          cout << queue.front() << " ";
58          queue.pop();
59      }
60      cout << endl;
61  }
```

실행 결과

```
기부 금액 범위 $00~09:  3  1  3  7  9  1  8  8  9  7  4  3
기부 금액 범위 $10~19:  18  14  13  15  16  15
기부 금액 범위 $20~29:  23  26  28  29  23  21  26  21  22  27  28  23
기부 금액 범위 $30~39:  32  31  39  39  39  34  30  38  30  36  32
기부 금액 범위 $40~49:  41  49  49  43  47  43  45  46  41
```

4 priority_queue 클래스

〈queue〉 헤더 파일에 정의되어 있는 priority_queue 클래스(우선 순위 큐)는 컨테이너 어댑터 클래스입니다. 우선 순위 큐는 요소를 삽입했을 때, 어떤 순위에 따라서 요소를 꺼낼 수 있습니다. 따라서 top 함수를 호출하면 가장 큰 요소부터 꺼내집니다.

연산

우선 순위 큐의 인터페이스는 [표 19-3]과 같습니다. 큐와 마찬가지로 push 함수로 요소를 뒤에 넣고, pop 함수로 앞의 요소를 꺼냅니다. 다만 접근할 때는 앞의 요소를 꺼내는 top 함수만 사용할 수 있습니다. 뒤의 요소는 접근할 수 없으므로 주의하세요.

큐와 우선 순위 큐의 큰 차이는 우선 순위 큐는 관계 연산자를 지원하지 않는다는 것입니다([표 19-3]). 따라서 두 우선 순위 큐를 비교하는 작업을 할 수 없습니다.

활용

우선 순위 큐는 우선 순위라는 특성과 큐의 특성을 모두 갖는 특수한 구조를 갖고 있습니다. 어떤 우선 순위를 가진 상태로 입력된 순서대로 요소를 꺼낼 때 활용합니다. 예를 들어 음식점은 손님이 들어온 순서로 자리를 배정합니다. 다만 예약한 손님이 있다면 예약 손님에게 우선 순위를 줍니다.

pair 클래스와 tuple 클래스, 사용자 정의 클래스 등을 활용해서 우선 순위 큐의 요소를 만들 수도 있습니다. 다만 항상 우선 순위를 나타낼 수 있는 함수와 함수 객체가 필요합니다. 이후에 함수 객체를 배우기 전까지 이와 관련된 내용은 설명을 미룹니다. [프로그램 19-14]는 우선 순위 큐를 활용해서 정수를 정렬하는 예입니다.

프로그램 19-14 우선 순위 큐 사용하기 Prg19-14.cpp

```
1  /*************************************************************
2   * priority_queue를 사용하는 프로그램                        *
3   *************************************************************/
4  #include <queue>
5  #include <iostream>
6  using namespace std;
7
8  int main()
9  {
10     // priority_queue 객체 생성
11     priority_queue<int> line;
12     // 요소 추가
13     line.push(4);
14     line.push(7);
15     line.push(2);
16     line.push(6);
17     line.push(7);
18     line.push(8);
```

```
19      line.push(2);
20      // 우선 순위에 따라 요소 출력
21      while(!line.empty())
22      {
23          cout << line.top() << "    ";
24          line.pop();
25      }
26      return 0;
27  }
```

실행 결과

```
8    7    7    6    4    2    2
```

우선 순위가 높은 요소가 우선 순위보다 낮은 요소보다 먼저 출력된다는 것을 확인할 수 있습니다.

05 연관 컨테이너

1 공용 인터페이스

연관 컨테이너associative container는 키key를 기반으로 값value을 저장하고 접근합니다. 연관 컨테이너는 일반적으로 이진 탐색 트리로 구현되어 있습니다. 이번 장에서는 set 클래스와 map 클래스를 다룹니다. set 클래스는 키와 값이 동일한 연관 컨테이너입니다. map 클래스는 키와 값이 다른 연관 컨테이너입니다. 현재 표준에는 multiset과 multimap 클래스가 추가되었지만, 이와 관련된 내용은 이 책에서 다루지 않습니다.

연관 컨테이너를 하나하나 살펴보기 전에 [표 19-4]에서 공용 인터페이스를 살펴봅시다. AS는 set, map으로 대체해서 생각해주세요. 약자 S는 set, M는 map을 나타냅니다. 연관 컨테이너의 템플릿은 set 클래스의 경우 〈K〉, map 클래스의 경우 〈K, T〉를 사용합니다. K는 키의 자료형을 의미하고, T는 값의 자료형을 의미합니다.

표 19-4 연관 컨테이너의 공용 인터페이스

생성자, 할당 연산자, 소멸자	S	M
AS〈...〉::AS()	V	V
AS〈...〉::AS(const_iterator first, const_iterator last)	V	V
AS〈...〉::AS(const AS〈...〉& s)	V	V
AS〈...〉&AS〈...〉::operator=(const AS〈...〉& right)	V	V
AS〈...〉::~AS()	V	V
크기 조절	**S**	**M**
size_type AS〈...〉::size()	V	V
size_type AS〈...〉::max_size()	V	V
bool AS〈...〉::empty()	V	V
반복자	**S**	**M**
iterator AS〈...〉::begin()	V	V
iterator AS〈...〉::end()	V	V
reverse_iterator AS〈...〉::rbegin()	V	V

reverse_iterator AS<...>::rend()	V	V
접근	**S**	**M**
T& AS<...>::operator[](const K& k) const		V
탐색	**S**	**M**
size_type AS<...>::count(const K& k) const	V	V
iterator AS<...>::find(const K& k) const	V	V
iterator AS<...>::lower_bound(const K& k) const	V	V
iterator AS<...>::upper_bound(const K& k) const	V	V
pair<iterator, iterator> AS<...>::equal_range(const K& k) const	V	V
삽입	**S**	**M**
pair<iterator, bool> AS<...>::insert(const K& k)	V	
pair<iterator, bool> AS<...>::insert(const pair <const K, T>& p)		V
iterator AS<...>::insert(iterator hintpos, const K& element)	V	V
void AS<...>::insert(InputIterator first, InputIterator last)	V	V
제거	**S**	**M**
size_type AS<...>::erase(const K& k)	V	V
void AS<...>::erase(iterator pos)	V	V
void AS<...>::erase(iterator first, iterator last)	V	V
void AS<...>::clear()	V	V
스왑	**S**	**M**
void AS<...>::swap(AS<...>& v)	V	V
전역 함수	**S**	**M**
bool operator oper(const AS<...>left, const AS<...>right)	V	V
void swap(AS<...>& c1, AS<...>& c2)	V	V

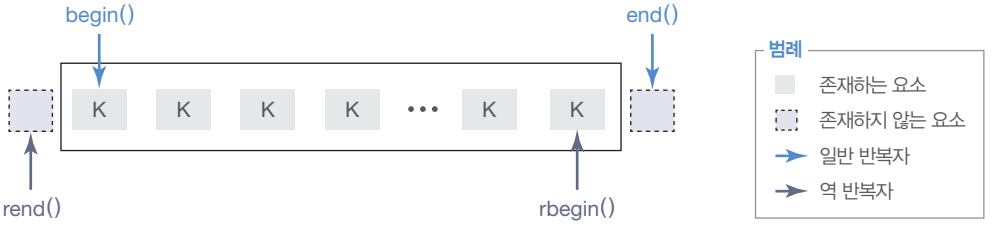
set 클래스와 map 클래스를 살펴보기 전에 〈utility〉 헤더 내부에 정의된 pair라는 구조체에 대해서 살펴봅시다. pair는 2개의 템플릿 데이터 멤버로 정의됩니다. 이때 두 데이터 멤버의 자료형은 다를 수 있습니다. 코드로 살펴보면 다음과 같습니다.

```
template<typename T1, typename T2>
struct pair
{
    T1 first;
```

```
    T2 second;
};
```

2 set 클래스

set 클래스(세트, 집합)는 〈set〉 헤더 파일에 정의되어 있습니다. 세트는 키key라고 부르는 하나의 템플릿 자료형만 지정합니다. 키는 오름차순으로만 정렬되며, 중복은 허용되지 않습니다. [그림 19-11]은 세트를 나타낸 그림입니다.



- set 클래스의 요소는 유일성을 가짐(중복 허용 불가)
  각각의 요소는 비선형적인 관계를 가짐
  set 클래스는 양방향 반복자를 지원

그림 19-11 set 클래스

그림은 세트를 선형적으로 그렸지만, 실제로 내부적으로 세트는 비선형적으로 구성되어 있습니다. 일반적으로는 이진 탐색 트리를 사용해 구현합니다.

연산

세트를 사용할 때 활용할 수 있는 연산을 간단하게 살펴봅니다.

■ 생성자, 소멸자, 할당 연산자

기본 생성자는 빈 세트를 생성합니다. 매개변수가 있는 생성자를 사용하면, 다른 세트에 있는 pos1부터 pos2까지의 데이터를 기반으로 세트를 생성할 수 있습니다(pos2 위치 제외). 복사 생성자 또는 할당 생성자를 사용해서도 세트를 만들거나 초기화할 수 있습니다.

```
set<type> set1                  // 빈 세트 생성
set<type>(pos1, pos2) set2      // 다른 세트의 일부 요소를 기반으로 세트 생성
set<type>(set2) set3            // 다른 세트의 요소를 기반으로 세트 생성
set4 = set3                     // 다른 세트의 요소를 기반으로 세트 생성
```

■ **크기 조절**

이전에 시퀀스 컨테이너를 살펴볼 때 보았던 size, max_size, empty 함수를 사용할 수 있습니다.

■ **이터레이터**

세트는 임의 반복자가 아니라 양방향 반복자를 사용합니다. 시퀀스 컨테이너처럼 4개의 상수 반복자, 4개의 비상수 반복자를 제공합니다.

```
set1.begin()          // 첫 번째 요소를 가리키는 기본 반복자
set1.end()            // 마지막 요소 뒤를 가리키는 기본 반복자
set1.rbegin()         // 마지막 요소를 가리키는 역 반복자
set1.rend()           // 첫 요소 앞을 가리키는 역 반복자
```

■ **탐색**

세트 내부의 요소들은 정렬되어 있으므로, 굉장히 효율적으로 탐색할 수 있습니다. 탐색을 할 때는 다음과 같은 5개의 멤버 함수를 사용합니다.

```
set1.count(k)         // k 값의 수를 리턴
set1.find(k)          // k의 위치를 가리키는 반복자를 리턴
set1.lower_bound(k)   // k를 삽입할 수 있는 첫 위치 리턴
set1.upper_bound(k)   // k를 삽입할 수 있는 마지막 위치 리턴
set1.equal_range(k)   // lower_bound와 upper_bound 쌍을 리턴
```

세트 내부의 요소는 중첩이 없으므로 count(k)는 0 또는 1만 리턴합니다. find 멤버 함수는 세트에서 값을 찾고, 값을 찾을 경우 반복자를 리턴합니다. 만약 값을 찾지 못했다면 end 반복자를 리턴합니다. lower_bound 멤버 함수는 요소를 삽입할 수 있는 첫 위치를 리턴하며, upper_bound 멤버 함수는 마지막 위치를 리턴합니다. equal_range 멤버 함수는 lower_bound와 upper_bound 쌍(pair 객체)을 리턴해줍니다. 참고로 세트에는 인덱스를 사용해서 요소에 접근할 수 있는 첨자 연산자와 at 함수가 없으므로 키를 사용해서 접근해야 합니다.

■ **삽입**

세트는 요소를 삽입할 때 push 함수가 아니라, insert 함수를 사용합니다. insert 함수는 키와 반복자를 사용해서 요소를 삽입합니다.

```
set1.insert(k)      // k를 삽입하고 반복자와 성공 여부를 나타내는 <pos, bool> 리턴
set1.insert(hint, k)      // k를 삽입하고 힌트 뒤에 추가된 반복자를 리턴
set1.insert(pos1, pos2)   // 다른 세트에 있는 요소 여러 개 삽입
```

첫 번째 형태는 키 k를 삽입하고, 위치를 나타내는 반복자와 성공 결과를 나타내는 불 값의 쌍(pair 객체)을 리턴합니다. 두 번째 형태는 k를 삽입합니다. 이때 탐색을 줄이고자 힌트가 되는 반복자를 지정한 형태입니다. 세 번째 형태는 다른 세트에 있는 pos1부터 pos2까지의 요소를 적절한 위치에 삽입합니다(pos2 위치 제외).

■ **제거**

세트는 요소를 제거할 때 pop 함수가 아니라 erase 함수를 사용합니다. erase 함수는 키와 반복자를 사용해서 요소를 제거합니다.

```
set1.erase(k)      // k를 제거하고 반복자와 성공 여부를 나타내는 <pos, bool> 리턴
set1.erase(pos)          // pos 위치의 요소를 제거
set1.erase(first, last)  // (first, last) 사이의 요소 제거
set1.clear()             // 모든 요소를 제거한 뒤 반복자 리턴
```

첫 번째 형태는 키 k를 사용해서 요소를 제거하고, 요소 수를 리턴합니다. 세트는 요소 중첩이 없으므로, 수는 0 또는 1입니다. 두 번째 형태는 pos에 위치하는 요소를 제거합니다. 세 번째 형태는 범위로 요소를 제거하며, 마지막 네 번째 형태는 모든 요소를 제거합니다.

■ **이외의 연산자**

시퀀스 컨테이너처럼 세트에는 swap 함수와 비교 연산자가 있습니다.

활용

세트를 활용하는 예제 2개를 살펴봅시다. 첫 번째 예제는 정수를 오름차순과 내림차순으로 정렬하고, 그들의 정렬 관계를 출력하는 예제입니다. [프로그램 19-15]를 살펴보겠습니다.

프로그램 19-15 정수 세트 다루기 Prg19-15.cpp

```
1  /*************************************************************************
2   * 세트로 정수를 정렬하는 프로그램                                       *
3   *************************************************************************/
4  #include <set>
5  #include <iostream>
```

```
6  #include <iomanip>
7  using namespace std;
8
9  int main()
10 {
11     // 빈 정수 세트 생성
12     set<int> st;
13     // 중복을 포함하는 정수를 세트에 넣기
14     st.insert(47);
15     st.insert(18);
16     st.insert(12);
17     st.insert(24);
18     st.insert(52);
19     st.insert(20);
20     st.insert(24);
21     st.insert(92);
22     st.insert(53);
23     st.insert(77);
24     st.insert(98);
25     st.insert(87);
26     // 오름차순으로 세트의 요소 출력
27     cout << "오름차순으로 세트의 요소 출력하기" << endl;
28     set<int>::iterator iter;
29     for(iter = st.begin(); iter != st.end(); iter++)
30     {
31         cout << setw(4) << *iter;
32     }
33     cout << endl << endl;
34     // 내림차순으로 세트의 요소 출력
35     cout << "내림차순으로 세트의 요소 출력하기"<< endl;
36     set<int>::reverse_iterator riter;
37     for(riter = st.rbegin(); riter != st.rend(); riter++)
38     {
39         cout << setw(4) << *riter;
40     }
41     cout << endl << endl;
42     // 52 뒤의 요소 출력
43     set<int>::iterator iter1 = st.find(52);
44     iter1++;
45     cout << "52 뒤의 요소 = " << *iter1 << endl;
46     // 20 앞의 요소 출력
47     set<int>::iterator iter2 = st.find(20);
```

```
48      iter2--;
49      cout << "20 앞의 요소 = " << *iter2 << endl;
50      return 0;
51  }
```

실행 결과

```
오름차순으로 세트의 요소 출력하기
  12  18  20  24  47  52  53  77  87  92  98

내림차순으로 세트의 요소 출력하기
  98  92  87  77  53  52  47  24  20  18  12

52 뒤의 요소 = 53
20 앞의 요소 = 18
```

이어서 두 번째 프로그램은 Student 클래스의 세트를 만들고, Student 클래스에 〈 연산자를 오버로드해서 학생을 정렬한 뒤 출력하는 예시입니다. [프로그램 19-16]~[프로그램 19-18]을 살펴보겠습니다. [프로그램 19-16]은 Student 클래스의 인터페이스 파일입니다.

프로그램 19-16 인터페이스 파일(student.h) Prg19-16.cpp

```
1   /*****************************************************************************
2    * Student 클래스의 인터페이스 파일                                          *
3    *****************************************************************************/
4   #ifndef STUDENT_H
5   #define STUDENT_H
6   #include <string>
7   #include <set>
8   #include <iostream>
9   #include <iomanip>
10  using namespace std;
11
12  class Student
13  {
14      private:
15          int identity;
16          string name;
17          double gpa;
18      public:
19          Student(int identity, string name, double gpa);
20          ~Student();
```

```
21          void print() const;
22          bool friend operator<(const Student& left, const Student& right);
23  };
24  #endif
```

[프로그램 19-17]은 Student 클래스의 구현 파일입니다.

프로그램 19-17 구현 파일(student.cpp) Prg19-17.cpp

```
1   /*************************************************************************
2    * Student 클래스의 구현 파일                                            *
3    *************************************************************************/
4   #include "student.h"
5
6   // 생성자
7   Student::Student(int id, string nm, double gp)
8   : identity(id), name(nm), gpa(gp)
9   {
10  }
11  // 소멸자
12  Student::~Student()
13  {
14  }
15  // print 멤버 함수
16  void Student::print() const
17  {
18      cout << setw(3) << right << identity << "  ";
19      cout << setw(12) << left << name << "  ";
20      cout << setw(6) << right << showpoint << setprecision(3);
21      cout << gpa << "  " << endl;
22  }
23  // friend 연산자 오버로드
24  bool operator<(const Student& left, const Student& right)
25  {
26      return(left.identity < right.identity);
27  }
```

애플리케이션 파일에서는 Student 객체를 인스턴스화하고, 해당 객체를 identity에 따라 정렬되게 세트에 삽입합니다. 이후에 각 객체의 정보를 출력합니다. [프로그램 19-18]을 살펴보겠습니다.

프로그램 19-18 애플리케이션 파일(app.cpp) Prg19-18.cpp

```
/*************************************************************************
 * Student 클래스를 사용하는 애플리케이션 파일                           *
 *************************************************************************/
#include "student.h"

int main()
{
    // Student 클래스의 인스턴스를 6개 생성
    Student student1(120, "George", 3.78);
    Student student2(185, "Mary", 3.95);
    Student student3(110, "Richard", 4.00);
    Student student4(245, "Alen", 3.70);
    Student student5(172, "John", 3.00);
    Student student6(195, "Lucie", 3.80);
    // 위의 인스턴스를 세트에 넣음
    set<Student> stdSet;
    stdSet.insert(student1);
    stdSet.insert(student2);
    stdSet.insert(student3);
    stdSet.insert(student4);
    stdSet.insert(student5);
    stdSet.insert(student6);
    // 출력
    set<Student>::iterator iter;
    for(iter = stdSet.begin(); iter != stdSet.end(); iter++)
    {
        iter->print();
    }
    return 0;
}
```

실행 결과

```
110  Richard  4.00
120  George  3.78
172  John  3.00
185  Mary  3.95
195  Lucie  3.80
245  Alen  3.70
```

3 map 클래스

map 클래스(맵)는 테이블table, 딕셔너리dictionary, 연관 배열associate array이라고도 불리는 객체로, 〈map〉 헤더에 정의되어 있습니다. 맵은 키key와 값value을 쌍으로 갖는 요소를 여러 개 저장하는 컨테이너입니다. 요소들은 기본적으로 키를 기반으로 오름차순 정렬됩니다. 맵에서 키는 유일해야 합니다. [그림 19-12]는 맵의 예를 나타낸 것입니다.

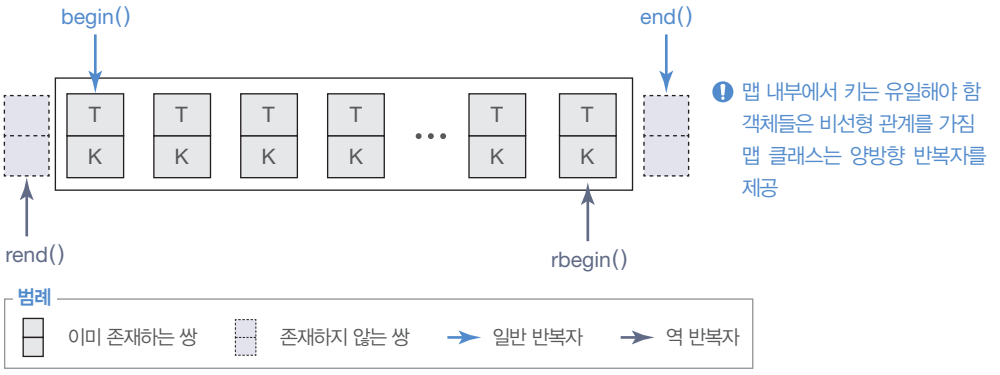


그림 19-12 맵

연산

맵은 세트와 굉장히 비슷합니다. 기본적인 차이는 맵은 요소에 접근할 때 [ ] 연산자를 사용한다는 것입니다. 연산자를 사용하므로 맵은 인덱스로 정수가 아니라 문자열을 사용하는 배열처럼 보이기도 합니다. 요소의 키를 알고 있으면 map[key] 같은 형태의 표현식으로 요소에 접근할 수 있습니다.

다만 [ ] 연산자가 벡터와 덱처럼 작용하지는 않습니다. 이는 단순하게 키-값 쌍을 나타내기 위한 표현 방법일 뿐입니다. 맵은 + 또는 - 연산자를 사용해서 요소들을 뛰어다닐 수 없으며, 양방향 반복자로만 반복을 돌 수 있습니다.

활용

맵을 사용하는 2가지 애플리케이션을 살펴봅시다. 첫 번째 애플리케이션에서는 학생의 이름과 점수 쌍으로 구성되는 요소를 갖는 맵을 만들고, 점수의 최대값과 최소값을 찾습니다. [프로그램 19-19]를 살펴봅니다.

프로그램 19-19 학생의 이름과 점수 저장하기 Prg19-19.cpp

```
1  /*************************************************************************
2   * 맵에 학생의 이름과 점수를 저장하는 프로그램                          *
3   *************************************************************************/
4  #include <map>
5  #include <iostream>
6  #include <iomanip>
7  #include <utility>
8  using namespace std;
9
10 int main()
11 {
12     // 맵과 반복자 선언
13     map<string, int> scores;
14     map<string, int>::iterator iter;
15     // 맵에 학생의 이름과 점수 저장
16     scores["John"] = 52;
17     scores["George"] = 71;
18     scores["Mary"] = 88;
19     scores["Lucie"] = 98;
20     scores["Robert"] = 77;
21     // 학생의 이름과 점수 정렬해서 출력
22     cout << "학생의 이름과 점수" << endl;
23     for(iter = scores.begin(); iter != scores.end(); iter++)
24     {
25         cout << setw(10) << left << iter->first << "  ";
26         cout << setw(4) << iter->second << endl;
27     }
28     return 0;
29 }
```

실행 결과

```
학생의 이름과 점수
George      71
John        52
Lucie       98
Mary        88
Robert      77
```

두 번째 프로그램은 문자열 내부에 있는 단어의 빈도를 찾는 프로그램입니다. 키를 단어, 값을 단어의 빈도로 같은 요소를 갖는 freq라는 맵 객체를 활용합니다. 다음 예제에서는 입력의 끝을 알릴 때 Ctrl+Z를 사용했습니다. [프로그램 19-20]을 살펴봅니다.

프로그램 19-20 단어 개수 세기 Prg19-20.cpp

```
/*************************************************************************
 * 문장 내부의 단어 수를 출력하는 프로그램                                *
 *************************************************************************/
#include <map>
#include <string>
#include <iomanip>
#include <iostream>
using namespace std;

int main()
{
    // 맵, 반복자, 문장 선언
    map<string, int> freq;
    map<string, int>::iterator iter;
    string word;
    // 문장 읽고 맵으로 빈도 배열 생성
    cout << "문장을 입력하세요: " << endl;
    while(cin >> word)
    {
        ++freq[word];
    }
    // 단어와 빈도 출력
    for(iter = freq.begin(); iter != freq.end(); iter++)
    {
        cout << left << setw(10) << iter->first << iter->second << endl;
    }
    return 0;
}
```

실행 결과

```
문장을 입력하세요:
we are in the world of this and that and this and that
^Z
and       3
are       1
in        1
```

```
of      1
that    2
the     1
this    2
we      1
world   1
```

정렬을 쉽게 확인할 수 있게 모든 키를 소문자로 넣었습니다. 결과를 보면 맵으로 단어의 빈도가 구성되는 것을 볼 수 있습니다. 이전에 살펴보았던 빈도 배열과 다르게 이번에는 배열의 인덱스가 정수가 아니라 문자열이라고 생각하면 됩니다. [그림 19-13]을 살펴봅니다.

그림 19-13 맵으로 빈도 배열 만들기

06 함수 사용

1 함수에 대한 포인터

알고리즘은 컨테이너의 요소들에 어떤 처리를 할 때 사용합니다. 알고리즘은 라이브러리에서 제공하는 것을 사용할 수도 있고, 우리가 직접 만들어서 사용할 수도 있습니다. 알고리즘은 함수 또는 함수 객체functor를 활용합니다. 따라서 알고리즘을 알아보기 전에 함수와 함수 객체를 살펴봅시다.

메모리에 저장되는 모든 엔티티는 주소를 갖습니다. 함수의 정의도 메모리 위에 저장되므로 함수도 어떤 주소를 갖습니다. 배열의 이름이 배열의 첫 번째 요소에 대한 포인터인 것처럼, 함수의 이름도 함수가 저장된 메모리 위치의 첫 번째 바이트를 가리키는 포인터입니다. [그림 19-14]는 배열의 이름과 함수의 이름이 어떤 식으로 메모리 위치를 가리키는지 나타냈습니다.

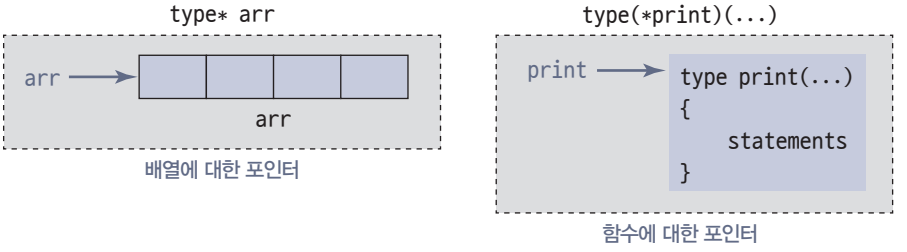


그림 19-14 배열에 대한 포인터와 함수에 대한 포인터

곧바로 예제를 살펴봅시다. [프로그램 19-21]은 함수를 매개변수로 받고 호출하는 예입니다. print 함수와 fun 함수를 정의했으며, fun 함수의 두 번째 매개변수로 print 함수에 대한 포인터를 전달해서 호출합니다.

프로그램 19-21 함수에 대한 포인터 사용하기 Prg19-21.cpp

```
1  /*************************************************************************
2   * 다른 함수를 호출하는 함수 사용하기                                    *
3   *************************************************************************/
4  #include <iostream>
5  using namespace std;
```

```
6
7  // print 함수의 정의
8  void print(int value)
9  {
10     cout << value << endl;
11 }
12 // fun 함수의 정의
13 void fun(int x, void(*f)(int))
14 {
15     f(x);
16 }
17
18 int main()
19 {
20     fun(24, print);       // fun 함수 호출
21     fun(88, print);       // fun 함수 호출
22     return 0;
23 }
```

실행 결과

```
24
88
```

[프로그램 19-21]은 함수에 대한 포인터를 살펴보기 위한 예제입니다. fun 함수의 선언에 있는 함수에 대한 포인터 void(*f)(int)가 어렵게 느껴질 수도 있지만, STL에 이미 fun 함수와 같은 함수들이 선언되어 있으므로, 이런 함수를 따로 직접 선언할 필요는 없습니다.

예를 들어서 [프로그램 19-22]는 STL에 정의되어 있는 제네릭 알고리즘 for_each를 사용한 것입니다. 반복자로 범위를 지정하고, 마지막 매개변수에 함수에 대한 포인터를 지정하기만 하면 범위 내부에 있는 요소들에 함수를 적용합니다. 참고로 이때 print 함수는 반복자가 가리키는 대상(*iterator)의 자료형을 매개변수로 받아야 합니다.

프로그램 19-22 사용자 정의 함수와 for_each 함수 조합해서 사용하기 Prg19-22.cpp

```
1  /*************************************************************************
2   * 범위 내의 모든 요소에 함수를 적용하는                                 *
3   * for_each 알고리즘을 사용하는 프로그램                                 *
4   *************************************************************************/
5  #include <vector>
6  #include <algorithm>
7  #include <iostream>
```

```
8  using namespace std;
9
10 // print 함수의 정의
11 void print(int value)
12 {
13     cout << value << "  ";
14 }
15
16 int main()
17 {
18     // 벡터 객체를 생성하고 요소 3개를 넣음
19     vector<int> vec;
20     vec.push_back(24);
21     vec.push_back(42);
22     vec.push_back(73);
23     // print 함수를 사용해서 요소 출력
24     for_each(vec.begin(), vec.end(), print);
25     return 0;
26 }
```

실행 결과

```
24  42  73
```

2 함수 객체

13장에서 함수 호출 연산자를 오버로드할 수 있다는 것을 알아보았습니다. 그리고 이를 활용하면 함수 객체functor를 만들 수 있다는 것도 예제로 알아보았습니다. [그림 19-15]는 기본적인 함수와 함수 객체를 만드는 방법을 나타낸 것입니다.

일반 함수

```
void sample(...)
{
    statements
}
```

함수 객체

```
class sample
{
    public :
        void operator()(...)
        {
          statements
        }
};
```

그림 19-15 일반 함수와 함수 객체를 만드는 방법

[프로그램 19-23]은 어떤 함수(main 함수)에서 함수 객체를 호출하는 방법을 나타낸 예입니다. 일단 Print 클래스를 만들고, operator()를 오버로드했습니다. 이어서 클래스의 인스턴스를 만들고 호출합니다. 객체 뒤에 괄호를 열고 매개변수를 넣으면, operator() 연산자 오버로드 함수를 호출하는 것입니다.

프로그램 19-23 함수 객체 사용하기 Prg19-23.cpp

```
1  /*************************************************************************
2   * 값을 출력하는 함수 객체 사용하기                                      *
3   *************************************************************************/
4  #include <iostream>
5  using namespace std;
6
7  class Print
8  {
9      public:
10     void operator()(int value) {cout << value;}
11 };
12
13 int main()
14 {
15     Print print;       // Print 객체 생성
16     print(45);         // operator() 호출
17     return 0;
18 }
```

실행 결과

```
45
```

함수 객체를 정의하는 코드가 일반적인 함수를 호출하는 코드보다 쓸데없이 길다고 느껴질 수 있습니다. 하지만 함수 객체는 다음과 같은 특징을 갖고 있습니다.

- 함수 객체는 객체를 사용할 수 있는 곳이라면 어디에서나 사용할 수 있습니다. 이를 활용하면 함수를 매개변수로 전달할 수 있게 하며, 함수를 리턴할 수 있게도 해줍니다.
- 함수 객체는 상태를 가질 수 있습니다. 따라서 호출마다 어떤 정보를 가질 수 있습니다.
- 함수 객체로 사용할 클래스는 상속을 활용해 다른 함수 객체를 추가로 파생할 수도 있습니다.

STL 알고리즘의 함수 객체

STL도 내부적으로 여러 함수 객체를 정의하고 있습니다. 이러한 함수 객체는 〈functional〉 헤

더 파일에 정의되어 있습니다. STL에서 정의하고 있는 함수 객체는 크게 단항, 이항 함수 객체로 구분할 수 있습니다.

표 19-5 STL의 함수 객체

함수 객체	종류	애러티	연산자
negate〈T〉	수학	단항	-
plus〈T〉	수학	이항	+
minus〈T〉	수학	이항	-
multiplies〈T〉	수학	이항	*
divides〈T〉	수학	이항	/
modulus〈T〉	수학	이항	%
equal_to〈T〉	관계	이항	==
not_equal_to〈T〉	관계	이항	!=
greater〈T〉	관계	이항	〉
greater_equal〈T〉	관계	이항	〉=
less〈T〉	관계	이항	〈
less_equal〈T〉	관계	이항	〈=
logical_not〈T〉	논리	단항	!
logical_and〈T〉	논리	이항	&&
logical_or〈T〉	논리	이항	||

이러한 함수 객체들은 내장 연산을 구현하고 있는 함수 객체입니다. 어떤 연산이 어떤 이름을 갖고 있는지 주의해서 생각해야 합니다. 예를 들어서 negate는 음수 연산자(-)를 구현하고 있으며, minus는 빼기 연산자(-)를 나타냅니다.

알고리즘과 관련된 내용은 다음 장에서 설명합니다. 이번 절에서는 [프로그램 19-24]처럼 transform 알고리즘을 사용해서 모든 요소의 부호를 바꾸는 예제만 살펴봅니다. 코드를 보면 negate 객체를 생성하는 부분까지만 있습니다. 이는 transform 함수에서 내부적으로 negate 객체를 호출하기 때문입니다.

프로그램 19-24 단항 함수 객체 Prg19-24.cpp

```
/*************************************************************************
 * 함수에 대한 포인터와 함수 객체를 사용하는 프로그램                    *
 *************************************************************************/
#include <vector>
#include <algorithm>
#include <iostream>
#include <functional>
using namespace std;

// 사용자 정의 print 함수
void print(int value)
{
    cout << value << " ";
}

int main()
{
    // 4개의 요소를 갖는 벡터 생성
    vector<int> vec;
    vec.push_back(24);
    vec.push_back(42);
    vec.push_back(73);
    vec.push_back(92);
    // 함수에 대한 포인터로 요소 출력
    for_each(vec.begin(), vec.end(), print);
    cout << endl;
    // 모든 요소의 부호를 반전하고 출력
    transform(vec.begin(), vec.end(), vec.begin(), negate<int>());
    for_each(vec.begin(), vec.end(), print);
    return 0;
}
```

실행 결과

```
24 42 73 92
-24 -42 -73 -92
```

07 알고리즘

1 소개

STL의 마지막 구성 요소는 알고리즘입니다. 알고리즘은 각각의 컨테이너 내부에 어떤 연산을 따로 정의하지 않고, 반복자를 사용해서 연산을 처리할 수 있게 해주는 함수로써 모든 컨테이너에 적용할 수 있는 전역 함수입니다. 알고리즘은 기능에 따라서 비변경, 변경, 정렬, 수치 알고리즘으로 구분할 수 있습니다.

이때 컨테이너가 적절한 반복자를 지원하지 않는다면, 알고리즘을 적용할 수 없다는 것이 중요합니다. 예를 들어서 정렬 알고리즘은 임의 접근 반복자가 필요합니다. 하지만 list〈T〉 컨테이너는 임의 접근 반복자를 지원하지 않습니다. 따라서 list〈T〉에는 정렬 알고리즘을 적용할 수 없습니다. 그래서 list〈T〉는 내부적으로 별도의 멤버 함수로 정렬을 정의하고 있는 것입니다. 추가적으로 컨테이너 어댑터도 반복자를 지원하지 않으므로, 알고리즘을 적용할 수 없습니다. 알고리즘이 굉장히 많아서 알고리즘을 설명하는 것만으로도 책 한 권이 나오므로, 여기서는 몇 가지만 살펴봅니다.

2 비변경 알고리즘

〈algorithm〉 헤더 파일에 정의되어 있는 비변경 알고리즘non-mutating algorithm은 컨테이너의 순서를 변경하지 않습니다. 비변경 알고리즘은 요소를 변경할 수도 있고, 요소를 변경하지 않을 수도 있습니다. 다음은 많이 사용되는 비변경 알고리즘입니다. InIter는 Input Iterator(입력 반복자)의 약자입니다. Predicate pred는 불 값을 리턴하는 함수입니다.

역자 비변경 알고리즘은 컨테이너의 순서처럼 구조를 변경하지 않는다는 것이지, 요소는 변경할 수 있다는 점을 주의하기 바랍니다.

```
difference_type count(InIter first, InIter last, const T& value);
difference_type count_if(InIter first, InIter last, Predicate pred);
InIter find(InIter first, InIter last, const T& value);
Function for_each(InIter first, outIter last, Function func);
```

for_each 함수는 매개변수로 전달하는 Function f가 무언가를 리턴하면 요소를 변경합니다. 반대로 값을 리턴하지 않으면 요소를 변경하지 않습니다. count 함수는 매개변수로 전달한 값과 동일한 요소 수를 세고, count_if 함수는 매개변수로 전달한 pred 함수가 true를 리턴하는 경우의 수를 셉니다. find 함수는 매개변수로 전달한 값의 위치를 찾고 for_each 함수는 요소에 매개변수로 전달한 함수 func를 적용합니다. 모든 함수는 반복자 first 위치부터 last 위치 사이에 적용됩니다. [프로그램 19-25]는 정수 벡터에 for_each, count, count_if 알고리즘을 적용하는 예시입니다.

프로그램 19-25 비변경 알고리즘 사용하기 Prg19-25.cpp

```
1  /*************************************************************************
2   * 비변경 알고리즘을 사용하는 프로그램                                   *
3   *************************************************************************/
4  #include <vector>
5  #include <algorithm>
6  #include <iostream>
7  using namespace std;
8
9  // isEven 함수 정의
10 bool isEven(int value)
11 {
12     return(value % 2 == 0);
13 }
14 // timesTwo 함수 정의
15 void timesTwo(int& value)
16 {
17     value = value * 2;
18 }
19 // print 함수 정의
20 void print(int value)
21 {
22     cout << value << "  ";
23 }
24
25 int main()
26 {
27     // 정수의 벡터 생성
28     vector<int> vec;
29     // 벡터에 요소 10개 넣기
30     vec.push_back(17);
31     vec.push_back(10);
```

```
32      vec.push_back(13);
33      vec.push_back(13);
34      vec.push_back(18);
35      vec.push_back(15);
36      vec.push_back(17);
37      vec.push_back(13);
38      vec.push_back(13);
39      vec.push_back(18);
40      // 원본 출력
41      cout << "원본 벡터의 값" << endl;
42      for_each(vec.begin(), vec.end(), print);
43      cout << endl << endl;
44      // 벡터 내부의 10의 개수 세기
45      cout << "벡터 내부에 있는 10의 개수 = ";
46      cout << count(vec.begin(), vec.end(), 10);
47      cout << endl << endl;
48      // 벡터 내부의 짝수 개수 세기
49      cout << "벡터 내부에 있는 짝수의 개수 = ";
50      cout << count_if(vec.begin(), vec.end(), isEven);
51      cout << endl << endl;
52      // 벡터 내부에 있는 값 2배로 만들기
53      cout << "벡터 내부의 요소에 2 곱하기" << endl;
54      for_each(vec.begin(), vec.end(), timesTwo);
55      for_each(vec.begin(), vec.end(), print);
56      return 0;
57  }
```

실행 결과

```
원본 벡터의 값
17  10  13  13  18  15  17  13  13  18

벡터 내부에 있는 10의 개수 = 1

벡터 내부에 있는 짝수의 개수 = 3

벡터 내부의 요소에 2 곱하기
34  20  26  26  36  30  34  26  26  36
```

3 변경 알고리즘

〈algorithm〉 헤더 파일에 정의되어 있는 변경 알고리즘mutating algorithm은 컨테이너의 구조를 변경합니다. 굉장히 많은 알고리즘이 있지만, 다음 코드만 알아봅니다. BdIter는 양방향 반복자, FwIter는 전방 반복자를 의미합니다.

```
void generate(BdIter first, BdIter last, gen);
void reverse(BdIter first, BdIter last);
void rotate(FwIter first, FwIter middle, FwIter last);
void random_shuffle(BdIter first, BdIter last);
outIter transform(inIter first, inIter second, outIter start, oper);
```

generate 함수는 시퀀스를 생성하고 reverse 함수는 컨테이너의 순서를 반전합니다. rotate 함수는 첫 번째 요소를 마지막으로 옮겨서 컨테이너를 회전하고 random_shuffle 함수는 임의의 순서로 요소를 정렬합니다. transform 함수는 first와 second 사이의 요소에 oper 함수를 적용하고, 리턴된 결과를 start 위치부터 씁니다. 최종적으로 마지막 변경된 요소에 대한 포인터를 리턴합니다. [프로그램 19-26]은 3가지 변경 알고리즘을 사용하는 예입니다. 정수로 구성된 벡터를 만들고, 반전하고, 회전하고, 셔플합니다.

프로그램 19-26 변경 알고리즘 사용하기 Prg19-26.cpp

```
1  /*************************************************************************
2   * 변경 알고리즘을 사용하는 프로그램                                     *
3   *************************************************************************/
4  #include <vector>
5  #include <algorithm>
6  #include <iostream>
7  #include <iomanip>
8  using namespace std;
9
10 // print 함수의 정의
11 void print(int value)
12 {
13     cout << value << "  ";
14 }
15
16 int main()
17 {
18     // 벡터 인스턴스화
```

```
19      vector<int> vec;
20      // 요소 추가
21      vec.push_back(11);
22      vec.push_back(14);
23      vec.push_back(17);
24      vec.push_back(23);
25      vec.push_back(35);
26      vec.push_back(52);
27      // 원본 출력
28      cout << "원본 벡터" << endl;
29      for_each(vec.begin(), vec.end(), print);
30      cout << endl << endl;
31      // 벡터를 반전하고 출력
32      cout << "반전한 벡터" << endl;
33      reverse(vec.begin(), vec.end());
34      for_each(vec.begin(), vec.end(), print);
35      cout << endl << endl;
36      // 벡터를 회전하고 출력
37      cout << "회전한 벡터" << endl;
38      rotate(vec.begin(), vec.begin() + 2, vec.end());
39      for_each(vec.begin(), vec.end(), print);
40      cout << endl << endl;
41      // 벡터를 셔플하고 출력
42      cout << "셔플한 벡터" << endl;
43      random_shuffle(vec.begin(), vec.end());
44      for_each(vec.begin(), vec.end(), print);
45      cout << endl << endl;
46      return 0;
47  }
```

실행 결과

```
원본 벡터
11  14  17  23  35  52

반전한 벡터
52  35  23  17  14  11

회전한 벡터
23  17  14  11  52  35

셔플한 벡터
23  14  35  52  17  11
```

4 정렬 알고리즘

〈algorithm〉 헤더 파일에는 정렬 또는 정렬과 관련된 알고리즘을 제공합니다. 다음과 같은 알고리즘만 살펴봅시다.

```
void sort(RndIter first, RndIter last);
bool binary_search(FwIter first, FwIter last, const T& value);
FwIter min_element(FwIter first, FwIter last);
FwIter max_element(FwIter first, FwIter last);
OutIter set_difference(InIter first1, InIter last1, InIter first2,
                                    InIter last2, OutIter result);
OutIter set_intersection(InIter first1, InIter last2, InIter first2,
                                      InIter last2, OutIter result);
OutIter set_union(InIter first1, InIter last1, InIter first2, InIter last2,
                                                          OutIter result);
OutIter set_symmetric_difference(InIter first1, InIter last1, InIter first2,
                                               InIter last2, OutIter result);
```

sort 함수는 정렬할 때 사용합니다. 이때 시퀀스가 반드시 임의 접근 반복자를 지원해야 합니다. binary_search 함수는 이진 탐색 알고리즘을 사용해 요소를 찾습니다. min_element 함수는 가장 작은 요소를 찾고, max_element 함수는 가장 큰 요소를 찾습니다. 이어지는 4개의 알고리즘은 집합 연산으로 순서대로 차집합, 교집합, 합집합, 대칭 차집합을 나타냅니다.

[프로그램 19-27]은 정수 벡터를 만들고 오름차순과 내림차순으로 정렬합니다. 오름차순으로 정렬할 때는 less〈T〉 객체를 사용합니다. 기본적으로 오름차순 정렬되므로 별도로 입력하지 않아도 괜찮습니다. 내림차순으로 정렬할 때는 greater〈T〉 객체를 사용합니다.

프로그램 19-27 정렬 알고리즘 사용하기 Prg19-27.cpp

```
1  /*************************************************************************
2   * 정렬 알고리즘으로 벡터를 정렬하는 프로그램                            *
3   *************************************************************************/
4  #include <vector>
5  #include <algorithm>
6  #include <iostream>
7  using namespace std;
8
9  // print 함수의 정의
10 void print(int value)
11 {
```

```
12      cout << value << "  ";
13  }
14
15  int main()
16  {
17      // 벡터 인스턴스화
18      vector<int> vec;
19      // 요소 6개 추가하고 출력
20      vec.push_back(17);
21      vec.push_back(10);
22      vec.push_back(13);
23      vec.push_back(18);
24      vec.push_back(15);
25      vec.push_back(11);
26      cout << "원본 벡터" << endl;
27      for_each(vec.begin(), vec.end(), print);
28      cout << endl << endl;
29      // 벡터를 오름차순으로 정렬하고 출력
30      cout << "오름차순으로 정렬한 벡터" << endl;
31      sort(vec.begin(), vec.end());
32      for_each(vec.begin(), vec.end(), print);
33      cout << endl << endl;
34      // 벡터를 내림차순으로 정렬하고 출력
35      cout << "내림차순으로 정렬한 벡터" << endl;
36      sort(vec.begin(), vec.end(), greater<int>());
37      for_each(vec.begin(), vec.end(), print);
38      cout << endl << endl;
39      return 0;
40  }
```

실행 결과

```
원본 벡터
17  10  13  18  15  11

오름차순으로 정렬한 벡터
10  11  13  15  17  18

내림차순으로 정렬한 벡터
18  17  15  13  11  10
```

그럼 이번에는 binary_search 알고리즘을 테스트해봅시다. 이 알고리즘은 정렬되어 있는 시퀀스에 적용할 수 있는 알고리즘입니다. [프로그램 19-28]은 정수로 구성된 벡터를 생성하고, 정렬한 다음, 두 값을 찾습니다. 한 값은 벡터 내에 있고, 다른 한 값은 벡터 내에 없습니다.

프로그램 19-28 이진 탐색 알고리즘 사용하기 Prg19-28.cpp

```
/*************************************************************
 * 벡터에 이진 탐색을 사용하는 프로그램                      *
 *************************************************************/
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
    // 벡터 인스턴스화
    vector<int> vec;
    // 요소 6개 추가
    vec.push_back(17);
    vec.push_back(10);
    vec.push_back(13);
    vec.push_back(18);
    vec.push_back(15);
    vec.push_back(11);
    // 벡터 정렬
    sort(vec.begin(), vec.end());
    // 벡터 탐색
    cout << "10 탐색 결과 = " << boolalpha;
    cout << binary_search(vec.begin(), vec.end(), 10) << endl;
    cout << "19 탐색 결과 = " << boolalpha;
    cout << binary_search(vec.begin(), vec.end(), 19) << endl;
    return 0;
}
```

실행 결과

```
10 탐색 결과 = true
19 탐색 결과 = false
```

[그림 19-16]은 first와 second에 합집합 연산을 적용한 결과를 나타낸 것입니다. 합집합 연산을 하면 first와 second의 모든 요소들을 포함하는 새로운 집합이 만들어집니다. [프로그램 19-29]는 실제로 이를 코드로 살펴보는 예입니다.

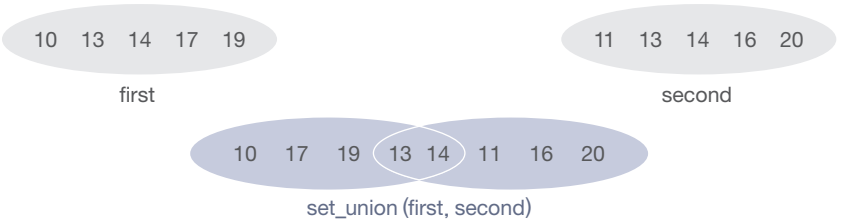


그림 19-16 set_union을 사용한 합집합

프로그램 19-29 두 집합의 합집합 구하기 Prg19-29.cpp

```
/*****************************************************************************
 * 합집합을 구하는 프로그램                                                  *
 *****************************************************************************/
#include <set>
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

// print 함수
void print(int value)
{
    cout << value << "  ";
}

int main()
{
    // 첫 번째 세트(집합) 생성
    set<int> first;
    first.insert(10);
    first.insert(19);
    first.insert(14);
    first.insert(17);
    first.insert(13);
    cout << "첫 번째 세트의 요소" << endl;
    for_each(first.begin(), first.end(), print);
    cout << endl << endl;
    // 두 번째 세트(집합) 생성
    set<int> second;
    second.insert(16);
    second.insert(14);
    second.insert(13);
```

```
33      second.insert(11);
34      second.insert(20);
35      cout << "두 번째 세트의 요소" << endl;
36      for_each(second.begin(), second.end(), print);
37      cout << endl << endl;
38      // 합집합 구하고 벡터에 저장
39      vector<int> temp(10);
40      vector<int>::iterator iter;
41      vector<int>::iterator endIter;
42      endIter = set_union(first.begin(), first.end(), second.begin(),
43                          second.end(), temp.begin());
44      // 벡터에서 결과 세트로 요소 복사
45      set<int> result;
46      for(iter = temp.begin(); iter != endIter; iter++)
47      {
48          result.insert(*iter);
49      }
50      cout << "결과 세트의 요소" << endl;
51      for_each(result.begin(), result.end(), print);
52      cout << endl << endl;
53      return 0;
54  }
```

실행 결과

```
첫 번째 세트의 요소
10  13  14  17  19

두 번째 세트의 요소
11  13  14  16  20

결과 세트의 요소
10  11  13  14  16  17  19  20
```

최종적인 결과를 내려면 시퀀스의 크기가 충분히 커야 합니다. 세트는 처음 생성될 때 비어있으므로, 세트만으로 구현할 수 없습니다. 따라서 결과를 일시적으로 저장하는 벡터를 만든 뒤, 이 내용을 세트에 복사해서 사용했습니다. 그림으로 정리해보면 [그림 19-17]과 같습니다.

그림 19-17 set_intersection 연산 사용하기

5 수치 알고리즘

〈numeric〉 헤더 파일에 정의되어 있는 수치 알고리즘numeric algorithm은 컨테이너의 요소 또는 컨테이너에 간단한 수학 연산을 적용해줍니다. 간단하게 다음의 알고리즘만 살펴봅니다.

```
T accumulate(InIter first, InIter last, T init)
```

accumulate 알고리즘은 (first, last) 범위에 있는 숫자의 합을 구하고, 이를 init과 더해서 리턴합니다. [프로그램 19-30]은 accumulate 알고리즘을 사용하는 예입니다.

프로그램 19-30 accumulate 알고리즘 사용하기 Prg19-30.cpp

```
1  /*****************************************************************************
2   * accumulate 수치 함수를 사용하는 프로그램                                  *
3   *****************************************************************************/
4  #include <vector>
5  #include <numeric>
6  #include <iostream>
7  #include <algorithm>
8  using namespace std;
9
10 // print 함수
11 void print(int value)
12 {
13     cout << value << "  ";
14 }
15
16 int main()
17 {
```

```
18      // 벡터 인스턴스화하고 출력
19      vector<int> vec;
20      vec.push_back(17);
21      vec.push_back(10);
22      vec.push_back(13);
23      vec.push_back(13);
24      vec.push_back(18);
25      vec.push_back(15);
26      vec.push_back(17);
27      for_each(vec.begin(), vec.end(), print);
28      cout << endl;
29      // 벡터 요소의 합을 구하고 출력
30      int sum = accumulate(vec.begin(), vec.end(), 0);
31      cout << "합계 = " << sum;
32      return 0;
33  }
```

실행 결과

```
17  10  13  13  18  15  17
합계 = 103
```

연습문제

01 모든 형태의 벡터를 매개변수로 받아서, 벡터의 요소를 출력하는 템플릿 함수를 작성하세요.

02 모든 형태의 리스트를 매개변수로 받아서, 리스트의 요소를 출력하는 템플릿 함수를 작성하세요.

03 10개의 요소를 갖는 벡터를 만들고, at 함수를 활용해 0, 1, 2, 3, 4, 5, 6, 7, 8, 9로 요소를 초기화하는 코드를 작성하세요.

04 0, 1, 2, 3, 4, 5, 6, 7, 8, 9를 요소로 갖는 리스트를 작성하는 코드를 작성하세요.

05 int 자료형을 요소로 갖는 벡터의 2번째 인덱스와 3번째 인덱스의 값을 스왑하는 코드를 작성하세요.

06 int 자료형을 요소로 갖는 벡터의 2번째 값과 3번째 값을 스왑하는 코드를 작성하세요. 임의 접근 반복자가 아닌, 일반 반복자와 ++ 연산자를 사용해서 스왑하는 코드를 작성하세요.

07 벡터에는 push_front 연산이 정의되어 있지 않습니다. int 자료형을 요소로 갖는 벡터의 가장 앞에 요소를 추가하는 코드를 직접 작성하세요.

08 벡터에는 pop_front 연산이 정의되어 있지 않습니다. int 자료형을 요소로 갖는 벡터의 첫 번째 요소를 제거하는 코드를 직접 작성하세요.

09 벡터에는 요소를 제거하는 함수가 정의되어 있지 않습니다. 벡터의 요소를 제거하는 함수를 직접 만드세요.

10 우선 순위 큐에 14, 24, 76, 18, 20, 54를 삽입한 후, 2개의 정수를 팝(pop)했다고 합시다. 이어서 정수를 팝하면, 어떤 정수가 팝 될지 예측하세요.

11 세트에 20, 17, 20, 14, 15, 19, 17, 10을 삽입한 후에 size 멤버 함수를 출력하면, 어떤 값이 출력될지 예측하세요.

12 맵에 (20, 10), (14, 40), (17, 3)이라는 페어가 들어있다고 합시다. 맵 내부에서 이러한 페어가 어떤 순서로 저장될지 예측하세요.

13 정수 20, 14, 18, 22, 76이 저장된 st라는 이름의 세트가 있다고 합시다. 이때 18과 22를 출력할 수 있는 코드를 작성하세요.

14 (15, 10), (16, 10), (6, 16), (12, 8)이라는 페어가 저장된 mp라는 이름의 맵이 있다고 합시다. 이때 다음 코드는 무엇을 출력할지 예측하세요.

```
cout << mp[6] << endl;
cout << mp[16] << endl;
cout << mp[10] << endl;
```

15 맵을 만들고 (3, 10), (5, 12), (7, 8)이라는 페어를 insert 멤버 함수를 활용해 삽입하는 코드를 작성하세요.

01 배열 포인터는 임의 접근 반복자 형태로 요소에 접근할 수 있으므로(서로 호환성이 있으므로), 제네릭 for_each 알고리즘을 배열에도 적용할 수 있습니다. 5개의 요소를 갖는 배열을 만들고, 출력하는 print 함수를 정의한 뒤, for_each 알고리즘을 활용해 배열의 요소를 출력하는 프로그램을 만드세요.

02 〈algorithm〉 헤더 파일에 정의되어 있는 max와 min 함수는 〈 연산자와 〉 연산자가 정의 되어 있는 두 객체의 크기를 비교할 때 활용할 수 있습니다. 이를 활용해서 두 int 자료형, 두 double 자료형, 두 string 자료형을 비교하는 간단한 프로그램을 만드세요. 참고로 max와 min 함수는 다음과 같은 템플릿 함수입니다.

```
const T& max(const T& first, const T& second)
const T& min(const T& first, const T& second)
```

03 이름 몇 개를 갖는 벡터를 우선 만듭니다. 이름을 출력하고, 정렬한 뒤, 정렬된 이름을 출력하는 프로그램을 만드세요.

04 리스트는 임의 접근 반복자를 지원하지 않으므로, 제네릭 sort 알고리즘을 사용할 수 없습니다. 그래서 STL 라이브러리는 리스트를 위해 별도의 sort 함수를 제공합니다. 정수 리스트를 만들고, sort 함수를 사용해 정렬하고 출력하는 프로그램을 만드세요.

05 배열 포인터는 임의 접근 반복자 형태로 요소에 접근할 수 있으므로(서로 호환성이 있으므로), 제네릭 sort 알고리즘을 배열에도 적용할 수 있습니다. 10개의 문자를 요소로 갖는 배열을 만들고, 정렬한 뒤 출력하는 프로그램을 만드세요.

06 10개의 요소를 갖는 벡터A와 빈 벡터B를 만들고, 벡터A의 2~6번째 위치의 요소를 벡터B로 복사한 뒤, 두 벡터의 요소를 모두 출력하는 프로그램을 만드세요.

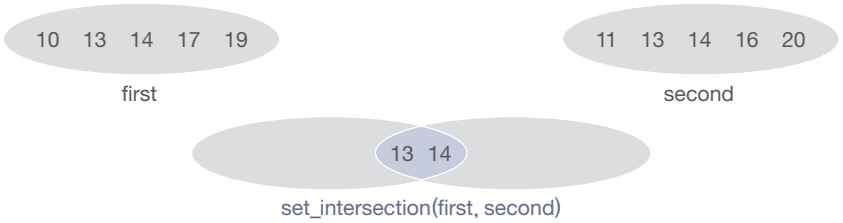
07 임의의 정수 10개를 갖는 벡터를 만들고, for_each와 accumulate 알고리즘을 활용해 값을 출력하고, 전체 합을 구하는 프로그램을 만드세요.

08 generate 알고리즘은 함수의 리턴값을 기반으로 컨테이너의 요소를 초기화하는 알고리즘입니다. randGen이라는 함수를 만든 후 이를 활용해서 100~200 범위의 랜덤한 정수 10개를 갖는 벡터를 만들고 출력하는 프로그램을 만드세요.

09 min_element와 max_element 알고리즘을 사용해서 벡터 내부의 요소 중 최소값과 최대값을 출력하는 프로그램을 만드세요.

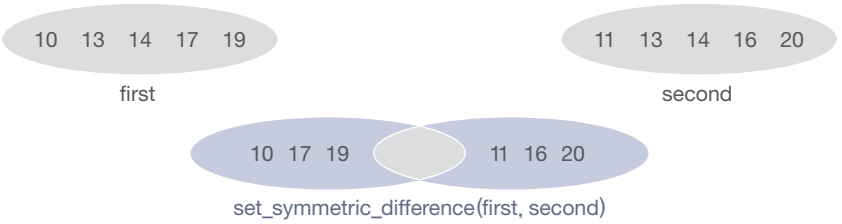
10 배열을 정렬하는 방법 중에 배열을 기반으로 세트를 만들고, 이를 다시 배열에 복사하는 방법이 있습니다. 이러한 형태로 배열을 정리하는 프로그램을 만드세요.

11 [프로그램 19-29]에서 합집합을 구해보았습니다. 이와 같은 형태로 교집합(intersection)을 구하는 프로그램을 만드세요. 교집합이란 다음 그림과 같이 두 집합 모두에 존재하는 요소를 구하는 연산입니다.



12 [프로그램 19-29]에서 합집합을 구해보았습니다. 이와 같은 형태로 차집합(difference)를 구하는 프로그램을 만드세요. 집합A와 집합B라는 2개의 집합이 있을 때, A - B와 B - A의 결과가 다릅니다. A - B는 A에만 있는 요소, B - A는 B에만 있는 요소를 구하는 연산입니다. 그림으로 나타내면 다음과 같습니다.

13 [프로그램 19-29]에서 합집합을 구해보았습니다. 이와 같은 형태로 대칭 차집합(symmetric difference)를 구하는 프로그램을 만드세요. 대칭 차집합이란 첫 번째 집합과 두 번째 집합에 소속되어 있지만, 두 집합 모두에 소속되지는 않은 요소만 구하는 연산입니다.



14 generate 알고리즘은 리스트의 이전 요소를 활용해 새로운 요소를 만들 때 사용할 수 있는 알고리즘입니다. 예를 들어서 이전 값에 2를 더하는 값을 리턴하는 형태의 Even이라는 함수 객체를 만들면, 이를 활용해 짝수 정수로 구성된 리스트를 만들 수 있습니다. 함수 호출 연산자를 오버로드하는 함수 객체를 만들고, 이를 활용해서 10개의 짝수로 구성되는 리스트를 generate 알고리즘으로 만드세요.

15 피보나치 수열은 함수 객체를 활용해 만들기 좋은 함수입니다. fib(n) = fib(n − 1) + fib(n − 2)로 구할 수 있습니다. 따라서 어떤 피보나치 수를 구하기 위해서는 이전 피보나치 수 2개를 저장하고 있어야 합니다. 함수 호출 연산자를 오버로드하는 Fib 클래스를 만들고, 이를 활용해서 fib(0)~fib(11)까지 11개의 요소를 갖는 벡터를 generate 알고리즘으로 만드세요.