고급 객체 지향 프로그래밍 제16장

강대기

May 28, 2008

1. C++ 표준 string 클래스
2. auto_ptr 템플릿
3. 표준 템플릿 라이브러리 (STL)
4. 일반화 프로그래밍
5. 함수 객체 (functor)
6. 알고리즘
7. 기타 라이브러리

string 클래스의 생성자들 (ctor)

- 생성자는 ctor, 파괴자는 dtor
- 1088쪽 표 16.1, 더 자세한 내용은 1349쪽 부록 F
- string은 템플릿 특수화인 basic_string<char>에 대한 typedef
- size_type 형은 string 헤더에 정의되어 있는 시스템마다 다른 정수형
- string::npos 는 문자열의 최대 길이 (아마도 79쪽의 climits 의 INT_MAX 와 동일)

string 클래스의 생성자들 (ctor)

1. C 스타일의 문자열
2. 개수와 문자
3. string 복사 생성자
4. 디폴트 생성자
5. C 스타일의 문자열과 정수 — 정수가 문자열 길이보다 커도 계속 복사
6. 템플릿 데이터형 매개 변수
   `template<class Iter> string(Iter begin, Iter end);`
   - string의 오버로딩된 [] 연산자의 주소는 매개 변수로 사용 가능 (1092쪽)

string 클래스의 기능들 (1091쪽)

- “+=” 오버로딩
- C 스타일 문자열 대입

C 스타일 문자열 입력

- C 스타일 문자열

```
char info[100];
cin >> info;              // 한 단어 읽음
cin.getline(info,100); // 한 행을 읽되, \n은 버림
cin.get(info,100);        // 한 행을 읽되, \n은 남겨놓음
```

  - cin >> info;는 ostream 클래스 메쏘드 — cin.operator>>(info);

string 입력

- string 객체의 경우

```
string stuff;
cin >> stuff;          // 한 단어 읽음
getline(cin, stuff); // 한 행을 읽되, \n은 버림
```

  - 자동 크기 조절 기능
  - `cin >> stuff;`는 일반 함수 — `operator>>(cin,stuff);`

string 입력을 위한 getline 함수

- 다음 중 어느 하나가 일어날 때까지 입력에서 문자들을 읽어 하나의 문자열로 저장
  - 파일의 끝을 만났을 때 — 입력 스트림의 eofbit이 설정됨 — fail()과 eof() 메쏘드가 true를 반환
  - 구분 문자(디폴트는 \n)에 도달했을 때 — 구분 문자는 입력 스트림으로부터 제거되고 저장되지 않음
  - 가능한 최대 문자수(string::npos 와 할당할 수 있는 메모리의 바이트 수 중 더 적은 것)를 읽었을 때 — 입력 스트림의 failbit가 설정됨 — fail() 메쏘드가 true를 반환
- 입력 스트림의 에러 상태
  - eofbit 설정 — 파일의 끝
  - failbit 설정 — 입력 에러
  - badbit 설정 — 하드웨어 실패와 같은 인식할 수 없는 실패
  - goodbit 설정 — 만사 OK

string 입력을 위한 operator>> 함수

- 구문 문자까지 읽고 버리는 대신, 화이트스페이스 문자까지 읽고 그 문자를 입력 큐에 남겨 둠
- 화이트스페이스 문자 — 빈 칸, 개행 문자 (\n), 리턴 문자 (\r), 수평 탭 문자, 수직 탭 문자, 폼 피드 등 좀 더 일반적으로 `isspace()`가 true를 반환하는 문자들

그 외의 문자열 작업

- 문자열 비교 — 여섯 개의 관계 연산자 모두 오버로딩 가능, string 객체와 C 문자열 비교 가능
- 문자열 크기 계산 — STL 호환성을 위한 `size()` 와 원래부터 있었던 `length()` (두 버전은 같은 일을 함)
- 부분 문자열 검색 — 1097 쪽의 표 16.2의 `find()` 함수들
  - `rfind()` — 가장 마지막으로 발생한 부분 문자열
  - `find_first_of()` — 전달 인자에 있는 문자들 중 가장 먼저 발생하는 문자
  - `find_last_of()` — 전달 인자에 있는 문자들 중 가장 나중에 발생하는 문자
  - `find_first_not_of()` — 전달 인자에 있는 문자들에 없으면서 가장 먼저 발생하는 문자
  - `find_last_not_of()` — 전달 인자에 있는 문자들에 없으면서 가장 나중에 발생하는 문자 (1363쪽)

그 밖의 기능 (부록 F)

- 문자열의 일부 또는 전체를 지우는 함수
- 문자열의 일부 또는 전체를 다른 문자열의 일부 또는 전체로 대체하는 함수 (1369쪽)
- 문자열의 중간에 어떤 내용을 삽입하거나 또는 문자열의 중간으로부터 어떤 내용을 삭제하는 함수 (1368쪽)
- 문자열의 일부 또는 전체를 다른 문자열의 일부 또는 전체와 비교하는 함수 (1364쪽)
- 문자열에서 부분 문자열을 추출하는 함수
- 문자열의 일부를 다른 문자열에 복사하는 함수 (1366쪽)
- 문자열의 내용을 서로 바꾸는 함수 (1370쪽)

자동 크기 조절 기능 외

- 이웃에 다른 사용 중인 메모리가 있을 수 있으므로, 새로 블럭을 할당해야 함 — 할당하는 크기는 현재 크기의 두 배
- `capacity()` — 현재 블록의 크기를 반환
- `reserve()` — 문자열 블록을 위한 최소 크기를 사용자가 요청
- `c_str()` — C 스타일 문자열의 포인터를 반환 (Java 의 `toString()`)
- 대소문자를 구분하지 않고 비교 — `stricmp()`
  - STL 의 `mismatch()` 와 `lexicographical_compare()` 가 더 좋음 (Effective STL의 항목 35)
- 실제 string 라이브러리들은 템플릿에 기초하고 있음 (`basic_string<char> string;`)

traits 템플릿 클래스

- char나 w_char 말고 다른 문자형이나 문자를 닮은 클래스에 대해서도 basic_string 템플릿을 사용 가능 — traits 템플릿 사용
- C++용 compile time type identification 기법
- Template Meta Programming 으로 가능
- 템플릿 특수화를 이용한 데이터 타입의 정의를 통한 사용 확장
- char과 w_char를 위한 char_traits 템플릿의 미리 정의된 특수화가 있음

```
template <> struct char_traits<char>
template <> struct char_traits<wchar_t>
```

Allocator 클래스

- basic_string 의 구조

```
template<class _Elem, class _Traits = char_traits<_Elem>,
  class _Ax = allocator<_Elem> >
class basic_string { };
```

- 메모리 할당을 책임을 지는 클래스
- 모든 STL 컨테이너는 프로그램이 사용하는 메모리 모델을 캡슐화하는 allocator를 사용함
- Allocator들은 포인터들의 크기, 메모리 구조, 재할당 모델 및 메모리 페이지 크기와 같이 플랫폼에 종속적인 세부들을 은닉함
- 컨테이너가 다른 할당자 종류들을 가지고 작동할 수 있기 때문에, 다른 할당자를 간단하게 그것 속으로 끼워 넣음으로써 다른 환경들에서 쉽게 작동할 수 있음
- 물론 구현할 때 모든 컨테이너를 위하여 적절한 할당자를 제공해야 함

auto_ptr 이 해결하고자 하는 문제

- 메모리 누설 (memory leak) - 1106쪽, 1107쪽 코드
- 함수가 종료되면 지역 변수인 포인터는 스택에서 해제되어 없어짐
- 객체의 경우는 파괴자가 수행되어 메모리 문제를 해결함
- 그렇다면, 포인터가 수명을 다하면 지시하는 메모리도 자동으로 해체될 수 없을까? → 템플릿을 통해 해결하자.

auto_ptr 사용

- auto_ptr 템플릿 → 주소를 담기 위한 포인터를 닮은 객체를 정의
- auto_ptr 객체의 수명이 다하면 파괴자가 delete를 이용하여 메모리를 해제
- 일반 포인터와 auto_ptr 의 차이 — 1108쪽

```
auto_ptr<double> pd(new double); // double 을 지시하는 auto_ptr
// double* 대신 사용
auto_ptr<string> ps(new string); // string 을 지시하는 auto_ptr
// string* 대신 사용
```

auto_ptr 사용

```
#include <memory>
void remodel(string& str)
{
  auto_ptr<string> ps(new string(str));
  ...
  // delete ps;  필요없음
  return;
}
```

- auto_ptr 생성자는 explicit 으로 암시적 데이터 형 변환이 허용되지 않음

```
auto_ptr<double> pd;
double* p_reg = new double;
pd = p_reg; // 허용되지 않는 암시적 변환
pd = auto_ptr<double>(p_reg); // 허용되는 명시적 변환
auto_ptr<double> pauto = p_reg; // 허용되지 않는 암시적 변환
auto_ptr<double> pauto(p_reg); // 허용되는 명시적 변환
```

auto_ptr 사용

- auto_ptr 는 부가적인 기능을 가지는 포인터의 역할을 하는 C++ 객체인 스마트 포인터 (smart pointer) 의 예
- 일반 포인터처럼 행동함
  - 내용 참조 (*ps), 증가 (ps++), 구조체 멤버에 접근 (ps->index), 일반 포인터에 대입 가능, 같은 데이터 형의 다른 auto_ptr 객체에 대입

auto_ptr 주의 사항

- auto_ptr 템플릿은 delete 를 사용하므로 auto_ptr<int> pi(new int[200]); 는 허용되지 않음
  - 동적 배열과 함께 사용할 수 있는 auto_ptr 대용물은 없음
  - memory 헤더 파일에서 auto_ptr 템플릿을 복사하여 auto_arr_ptr 로 수정한 후, delete 대신 delete[]를 사용할 수 있도록 수정할 수 있음
- auto_ptr 템플릿은 new 에 의해 할당된 메모리에만 사용해야 함

```
string vacation("나는 구름처럼 외롭게 방황했다.");
auto_ptr<string> pvac(&vacation);  // 허용되지 않음
```

auto_ptr 의 소유권 관리 전략은 파괴 복사 (destructive copy)

- 대입의 경우 소유권이 이전됨 – 소유권 개념이 없다면 두 개가 같은 string 객체를 지정하게 되어 delete를 두 번 발생하게 만듬

```
auto_ptr<string> ps(new string("나는 구름처럼 외롭게 방황했다."));
auto_ptr<string> vacation;
vacation = ps; // ps 가 널이 됨
```

- 복사의 경우에도 소유권이 이전됨 (1111 쪽)

```
auto_ptr<string> films(new string("개그 콘서트"));
auto_ptr<string> pwin(films);
```

- auto_ptr 의 컨테이너는 절대로 만들지 말자. (Effective STL 의 항목 8번) — Container of Auto Pointer (COAP) 절대 금지, 언급조차 금지

스마트 포인터 소유권 관리 전략 (Modern C++ Design)

1. 깊은 복사 또는 완전 복사 (deep copy)
   - 템플릿을 통해 객체를 복사할 경우, 기초 클래스 부분만 복사될 수 있음 — Java 처럼 clone() 함수가 해결책
2. Copy On Write (COW) — 고쳐질 때만 복사됨
   - 완벽한 전략이나, const 멤버함수와 non const 멤버 함수 호출을 구별할 수 없어 스마트 포인터 구현 어려움
3. 참조 카운팅 (reference counting)
4. 참조 연결 리스트 (reference linked list)
   - 순환 참조 (cycle reference) 문제 발생
5. 소유권 이전 또는 파괴 복사 (destructive copy)
   - 하나의 포인터만이 특정 객체를 소유

그 외의 스마트 포인터

- 참조 카운팅 (reference counting)
  - `boost::shared_ptr`, `boost::shared_array`
- 소유권 이전 또는 파괴 복사 (destructive copy)
  - C++ 표준으로 유일한 스마트 포인터인 `std::auto_ptr`

```
auto_ptr<_Ty>& operator=(auto_ptr<_Other>& _Right)// const 가 없음
```

    - 해볼만한 숙제 — `std::auto_ptr` 의 배열 버전
    - 한가지 대안: `auto_ptr< boost::array<...> >`
    - 쉽게 살려면, 아예 `vector`를 사용해도 됨
- 앞에서 언급 안된 여섯번째 전략 — 아예 복사를 막음
  - `boost::scoped_ptr`, `boost::scoped_array` (참고: `boost::noncopyable`)

표준 템플릿 라이브러리 (Alex Stepanov 와 Meng Lee)

- 일반화 프로그래밍 (generic programming) 이라는 패러다임을 제공함
  - 컨테이너 (container) — 배열, 큐, 리스트 등으로 동질적(homogeneous)임, 즉 같은 종류의 값들을 저장함
    - non-intrusive container — 컨테이너에 값의 복사본을 넣음
    - intrusive container — 컨테이너에 원래 값을 그대로 넣음 – STL에선 참조나 포인터로 구현됨
  - 이터레이터 (iterator) — 포인터의 일반화로 컨테이너 안에서 위치를 옮기는 것을 도와줌
  - 함수 객체 (function object; functor) — 함수 역할을 하는 객체로 상태를 가지는 함수를 표현할 수 있음
  - 알고리즘 (algorithm) — 배열을 정렬하거나 리스트의 특정 값을 검색, 또는 무작위화(randomize)

vector 템플릿 클래스

- 생성

```
#include <vector>
using namespace std;
vector<int> rating(5); // 5 개의 int 형 값을 가지는 벡터
int n;
cin >> n;
vector<double> scores(n); // n 개의 double 형 값을 가지는 벡터
```

- 대입, [] 연산자로 인덱싱

```
ratings[0]=9;
for (int i=0;i<n;i++) cout << scores[i] << endl;
```

- Allocator 객체를 지정하는 선택적 템플릿 전달 인자

```
template <class T, class Allocator = allocator<T> > class vector { ...
```

- 1114 쪽의 vect1.cpp

vector 가 컨테이너로서 가진 공통적인 메쏘드들

- 컨테이너 내의 원소들의 개수를 리턴하는 `size()`, 두 컨테이너의 내용을 교환하는 `swap()`, 컨테이너 내의 첫번째 원소를 참조하는 `begin()`, 컨테이너 내의 "마지막 원소 바로 다음(past-the-end)"을 참조하는 `end()`
  - 모든 컨테이너들이 이 메쏘드들을 가지고 있음
- 이터레이터는 포인터의 일반화로, 포인터이거나 포인터처럼 동작하는 객체일 수 있음 — 이터레이터를 통해 포인터를 일반화하여 다양한 컨테이너 클래스에 대해 일관된 인터페이스를 제공

이터레이터

- vector 컨테이너의 이터레이터는
  `vector<double>::iterator pd;`
- vector 객체에서 이터레이터 받기

```
vector<double>::iterator pd; // 이터레이터
vector<double> scores;
pd = scores.begin();
*pd = 22.3;
++pd;
```

- past-the-end — 마지막 원소 다음의 원소를 참조하는 이터레이터
  - C 스타일 문자열에서 마지막 실제 문자 다음의 원소가 널 문자라는 아이디어와 비슷

```
for (pd = scores.begin(); pd != scores.end() ; pd++)
  cout << *pd << endl;
```

push_back() 과 erase()

- push_back() — 벡터 끝에 원소 하나 추가
  - 필요한 경우, 자동으로 벡터의 크기가 늘어남

```
vector<double> scores; // 빈 벡터 생성
double temp;
while (cin >> temp && temp >= 0)
  scores.push_back(temp);
cout << scores.size() << " 개의 점수가 입력됨\n";
```

- erase() — 두 개의 이터레이터를 받아서 삭제
  - 첫번째는 삭제할 범위의 시작, 두번째는 삭제할 부분의 끝 바로 다음의 원소 — $[p1, p2)$

```
// 처음부터 두 개의 원소 삭제
scores.erase(scores.begin(), scores.begin()+2);
```

    - vector는 임의 접근을 제공하므로 scores.begin()+2 가 가능

insert()

- insert() — 세 개의 이터레이터 전달 인자들을 받아서 삽입
  - 첫번째 이터레이터는 새로원 원소들이 삽입될 위치 바로 앞의 위치
  - 두번째와 세번째 이터레이터는 삽입에 사용할 범위

```
vector<int> old;
vector<int> new;
...
old.insert(old.begin(), new.begin()+1, new.end());
...
old.insert(old.end(), new.begin()+1, new.end());
```

- 1119 쪽 프로그램 vect2.cpp

vector 에서 할 수 있는 그 밖의 것

- STL 방식에 따르면 검색, 정렬, 무작위화 등의 작업에 대해 각 객체에 대한 멤버 함수로 정의하지 않고, 컨테이터들에 대해 공통된 함수로 정의
  - 필요한 함수들의 개수가 줄어듬
- 대표적인 STL 함수들 — `for_each()`, `random_shuffle()`, `sort()` 등등

for_each(), random_suffle(), sort()

- for_each() — 첫번째와 두번째 전달 인자는 컨테이너의 범위를 지정, 세번째는 한 개의 전달 인자를 받는 함수 객체

```
vector<Review>::iterator pr;
for (pr=books.begin();pr!=books.end();pr++)
  ShowReview(*pr);
---> for_each(books.begin(), books.end(), ShowReview);
```

- random_suffle() — 첫번째와 두번째 전달 인자는 컨테이너의 범위를 지정

```
random_suffle(books.begin(), books.end());
```

- sort() — 첫번째와 두번째 전달 인자는 컨테이너의 범위를 지정, bool을 반환하는 객체에 대한 < 연산자(멤버 함수일수도 있고 일반 함수일수도 있음)를 이용하여 오름차 순으로 정렬

```
vector<int> coolstuff;
sort(coolstuff.begin(), coolstuff.end());
```

sort()

- sort() — 세번째 전달 인자로, 객체 두 개를 각각 전달 인자들로 받고 bool을 반환하는 함수 객체 (1168쪽) 사용 가능

```
bool WorseThan(const Review& r1, const Review& r2)
{
  if (r1.rating < r2.rating) return true; else return false;
}
...
sort(books.begin(), books.end(), WorseThan);
```

- 전체 순서화 (total ordering) - $a < b$ 이고 $b > a$ 이면, $a = b$ - a와 b는 동일하다. (identical)
- 순약 순서화 (strict weak ordering) - $a < b$ 이고 $b > a$ 이라고 해도, $a \neq b$ 일 수도 있음 - a와 b는 동등하다. (equivalent) — Effective STL 의 항목 19
- 1125 쪽 프로그램 vect3.cpp

일반화 프로그래밍

- STL 은 일반화 프로그래밍의 한 예
- 일반화 프로그래밍의 목적은 데이터 형과 무관한 코드를 작성하는 것
  - 템플릿은 데이터 형을 일반화하였음
  - STL은 알고리즘의 일반화된 표현을 제공함으로써 한 걸음 더 나아감
  - 이터레이터 — 알고리즘을 데이터 형과 무관하게 하기 위한 열쇠
- 비판적 시각 — “컨테이너에 독립적인 코드란 거는 환상” (Effective STL 항목 2)

이터레이터가 필요한 이유

- 템플릿만 사용해서는 일반화에 한계가 있음
  - `find_ar()` (1128 쪽) — double 을 가진 배열에서 데이터 찾기
  - `find_ll()` (1129 쪽) — double 을 가진 링크드 리스트에서 데이터 찾기
- 문제 정의 — 배열, 링크드 리스트, 그 밖의 다른 컨테이너형 데이터 구조들에 대해 하나의 find 함수를 가질 수 있을까?
- 이를 위한 이터레이터의 요건들
  - * 연산자로 내용 참조
  - 이터레이터끼리의 대입 가능
  - 이터레이터끼리의 비교 가능
  - 컨테이너의 모든 객체들을 훑고 지나가기 위한 ++ 연산자

앞의 함수들을 이터레이터 식으로 변환

- find_ar() — 1130쪽 이터레이터 typedef 도입, 1131쪽 전달 인자 받는 방법
- find_ll() — 1131쪽 이터레이터 객체 도입, 1132쪽 for 루프
- 최종적으로 링크드 리스트가 마지막 원소 바로 다음에 하나의 원소를 더 가지게 해서 두 방식을 통일 – (past-the-end 원소 또는 sentinel 값)
- 결과적인 코드 — 1133 쪽

```
vector<double>::iterator p;
또 는
list<double>::iterator p;

for (p=scores.begin();p!=scores.end();p++) cout<<*p<<endl;
```

앞의 함수들을 이터레이터 식으로 변환

- 좀 더 나아가 내부적인 처리까지 해주는 `for_each()` 사용
- 결국 내부적인 표현이 전혀 다른 컨테이너들에 대해서, 동일한 코드를 작성할 수 있음

이터레이터의 종류

- 알고리즘이 다르면 다른 이터레이터들이 요구됨
  - 검색 알고리즘 — ++ 연산자 요구, 데이터는 읽을 수 있으나 쓰기는 금지 (InputIterator)
  - 정렬 알고리즘 — + 연산자를 통한 임의 접근, 데이터는 읽고 쓸 수 있어야 함 (RandomAccessIterator)
- 입력 이터레이터 (InputIterator), 출력 이터레이터 (OutputIterator), 전방 이터레이터 (ForwardIterator), 전후방 이터레이터(BidirectionalIterator), 임의 접근 이터레이터(RandomAccessIterator) — 1138쪽 표 16.4

이터레이터의 종류

- find 함수의 원형은 이런 식

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last,
  const T& value);
```

- sort 함수의 원형은 이런 식

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

- 다섯 이터레이터 모두 내용 참조(* 연산자 추천), 비교 (==,!= 연산자 추천), 내용 비교 (==,!= 연산자 추천)
- 두 이터레이터가 `it1==it2` 이면 `*it1==*it2` 이어야 함

입력 이터레이터

- 컨테이너로부터 값을 읽기 위해 프로그램이 사용
- 내용은 참조할 수 있되, 값은 변경할 수 없음
- ++ 연산자로 모든 내용 참조 가능
- past-the-end에 도달하면 다 읽은 것으로 간주하고 멈춤
- 다시 한 번 읽었을 때, 같은 순서나 같은 내용이라는 보장 없음
- 이터레이터가 일단 증가된 후, 증가하기 전 값을 읽을 수 있다는 보장 없음 — 입력 이터레이터에 기반한 알고리즘은 일회성 알고리즘
- 단방향 이터레이터 — 증가시킬 수 있지만, 감소시킬 수 없음

출력 이터레이터

- 컨테이너로 값을 쓰기 위해 프로그램이 사용
- 내용을 참조할 수 없고 변경하는 것을 허용
- 나머지는 입력 이터레이터와 비슷함
- 출력 이터레이터에 기반한 알고리즘은 일회성 쓰기 전용 알고리즘

전방 이터레이터

- ++ 연산자로 모든 내용을 훑고 지나감
- 한 번에 한 원소 씩 전방으로 진행
- 다시 읽었을 때, 같은 순서로 훑고 지나감을 보장함
- 이터레이터가 증가된 후에도, 그 전의 내용을 참조할 수 있어, 다중 패스 알고리즘이 가능함
- 데이터를 읽고 변경할 때, 사용 가능함 (물론 데이터를 읽기만 할 때도 사용 가능)

전후방 이터레이터

- ++와 -- 연산자로 컨테이너 속을 전후방 모든 내용을 훑고 지나감
- 전방 이터레이터의 모든 기능에 후방 연산자 포함
- 예를 들어 reverse() 같은 함수에서 첫번째 원소와 마지막 원소를 바꿔나가는 과정 반복할 때 쓰임

임의 접근 이터레이터

- 표준 소팅이나 이진 탐색과 같은 알고리즘에서 임의의 원소로 접근하기 위한 기능이 필요함
- 포인터 덧셈/뺄셈 연산 (배열 인덱싱 포함), 원소들의 순서를 위한 관계 연산자
- 1137쪽 표 16.3

이터레이터 계층

- 이터레이터들은 기능적으로 계층을 형성함 (표 16.4)
  - 전방 이터레이터 = 입력 이터레이터 기능 + 출력 이터레이터 기능 + 자체 기능
  - 전후방 이터레이터 = 전방 이터레이터 기능 + 자체 기능
  - 임의 접근 이터레이터 = 전후방 이터레이터 기능 + 자체 기능
- 가장 제한 사항이 적은 이터레이터는 가장 넓은 범위의 알고리즘들에 사용 가능함
  - 입력 이터레이터를 위한 알고리즘에 임의 접근 이터레이터 사용 가능
  - 입력 이터레이터에 따라 제작된 find() 함수는 출력 이터레이터 말고 어떤 것이든 사용 가능
  - 임의 접근 이터레이터에 따라 제작된 sort() 함수는 임의 접근 이터레이터만 사용 가능

개념, 개량, 모델

- 이터레이터는 정의된 데이터 형이나 문법이 아닌 개념 — 원칙적으로 C++ 문법과 무관함
- 컴파일러가 전방 이터레이터의 특성을 가지는 클래스의 사용을 거기에 맞춰 제한할 수 없음
- STL의 개념들은 상속 관계와 비슷한 관계(개량, refinement라 부름)를 가지나 문법적인 상속 관계는 아님
  - 전후방 이터레이터는 전방 이터레이터의 개량
- 특정 개념의 한 구현을 모델이라 부름

이터레이터 자격의 포인터

- 이터레이터는 포인터의 일반화이므로, 포인터는 이터레이터임
- 따라서 STL 알고리즘은 포인터를 기초로 하는 컨테이너들에 대해서도 작동할 수 있음 — 예. 배열
- STL sort() 함수는 컨테이너에 든 첫번째 원소를 지시하는 이터레이터와 past-the-end 원소를 지시하는 이터레이터를 받는 데, 배열에 대해서도 적용 가능

```
const int SIZE =100;
double Receipts[SIZE];
sort(Receipts, Receipts+SIZE);
```

copy(), ostream_iterator, istream_iterator

- copy() 알고리즘은 하나의 컨테이너에서 다른 하나의 컨테이너로 데이터들을 복사
- 첫번째와 두번째 인자는 복사할 범위 지정 (입력 이터레이터) , 세번째 인자는 복사될 위치 (출력 이터레이터)

```
int casts[10] = { 6,7,2,9,4,11,8,7,10,5 };
vector<int> dice[10];
copy(casts, casts+10, dice.begin()); // 배열을 벡터에 복사
```

- 목적지 컨테이너의 기존 데이터 위에 덮어 써 버림 — 빈 벡터에는 데이터를 복사할 수 없음 — 이를 해결하기 위해서는 insert_iterator로 어댑터를 만듬 (1145 쪽)

ostream_iterator

- copy() 알고리즘으로 컨테이너에서 표준 출력으로 출력

```
#include <iterator>
...
ostream_iterator<int, char> out_iter(cout, " ");
```

- 첫번째 템플릿 전달인자는 출력 스트림으로 보내는 데이터 형, 두번째 템플릿 전달인자는 출력 스트림이 사용하는 데이터 형
- 첫번째 생성자 전달인자는 출력 스트림, 두번째 생성자 전달인자는 출력 스트림으로 보내진 각 항목 뒤에 표시되는 분리자

```
copy(dice.begin(), dice.end(), out_iter);
copy(dice.begin(), dice.end(), ostream_iterator<int,char>(cout," "));
```

istream_iterator

- copy() 알고리즘으로 표준 입력에서 컨테이너로 입력

```
copy(istream_iterator<int, char>(cin), istream_iterator<int, char>(),
  dice.begin());
```

- 첫번째 템플릿 전달인자는 입력 스트림에서 받을 데이터 형, 두번째 템플릿 전달인자는 입력 스트림이 사용하는 데이터 형
- 생성자 전달인자는 입력 스트림, 생략하면 입력이 실패했음을 의미
- 즉, 파일 끝, 데이터 형 불일치, 그외의 입력 실패가 일어날 때까지 입력 스트림으로부터 데이터를 읽는다는 의미

기타 유용한 이터레이터들

- iterator 헤더 파일이 정의하는 이터레이터들

  `istream_iterator`, `ostream_iterator`, `reverse_iterator`, `insert_iterator`, `back_insert_iterator`, `front_insert_iterator`
- `reverse_iterator`는 후진 출력함

  ```
  copy(dice.rbegin(), dice.rend(), out_iter);
  ```
- `rbegin()`는 past-the-end 위치이며, `rend()`는 첫번째 원소의 위치임
  - `rbegin()` 과 `end()`는 같은 값을 다른 형으로 리턴, `begin()` 과 `rend()` 도 마찬가지
  - 역방향 이터레이터는 먼저 감소시키고 내용 참조를 수행함
  - `rp`가 여섯번째 위치를 가리킨다면, `*rp` 는 다섯번째 위치의 값이 됨

copy()는 복사되는 컨테이너의 크기를 자동 조정 않함

- 삽입 이터레이터로 복사 과정을 삽입 과정으로 변환함으로써 해결
  - 다른 의견 : 삽입 이터레이터를 사용하는 copy()는 범위 지정 멤버 함수(예를 들어 insert() 멤버 함수)로 변환하면 훨씬 더 효율적임 (Effective STL 항목 5)
  - `back_insert_iterator` — 컨테이너 말미에 원소 삽입
    - 고정 시간($O(1)$)에 말미에 빠른 삽입을 허용하는 컨테이너형에만 사용 (예: vector,list,deque)
  - `front_insert_iterator` — 컨테이너의 선두에 원소 삽입
    - 고정 시간($O(1)$)에 선두에 빠른 삽입을 허용하는 컨테이너형에만 사용 (예: list,deque)
  - `insert_iterator` — 생성자에 전달 인자로 지정된 위치 앞에 원소 삽입
    - 위와 같은 제한이 없어 선두에 정보를 삽입할 수 있으나, `front_insert_iterator`에 비해 느림

이터레이터와 컨테이너

- 이 이터레이터들은 컨테이너 형을 템플릿 전달 인자로 사용하고 실제 컨테이너 식별자를 생성자 전달 인자로 사용함 — 1146쪽 `inserts.cpp`

```
back_insert_iterator<vector<int> > back_iter (dice);
insert_iterator<vector<int> > insert_iter (dice, dice.begin());
```

- vector
  - `empty()`, `size()`, `front()`, `back()`, `push_back()`, `pop_back()` 등등
- queue
  - `empty()`, `size()`, `front()`, `back()`, `push_back()`, `pop_front()` 등등
- copy()로 컨테이너에서 컨테이너로 복사, 컨테이너에서 출력 스트림으로 복사, 입력 스트림에서 컨테이너로 복사함

컨테이너의 종류

- 기본적으로 대부분의 경우, 컨테이너에는 객체가 복사(copy)되어 들어가고, 복사되어 나옴 (Effective STL 항목 3)
- 컨테이너 개념 — 컨테이너, 시퀀스 컨테이너, 결합 컨테이너
- 컨테이너 형 — 구체적인 컨테이너 객체들을 생성하는 데 사용할 수 있는 템플릿
  - deque, list, queue, priority_queue, stack, vector, map, multimap, set, multiset, bitset

컨테이너 형 정리 (Effective STL)

- 표준 STL 시퀀스 (sequence) 컨테이너 – deque, list (이중 링크드 리스트), vector, string
- 표준 STL 결합 (associative) 컨테이너 – map, multimap, set, multiset
- 비표준 시퀀스 컨테이너 – slist (싱글 링크드 리스트), rope (대용량 문자열)
  - `www.sgi.com/tech/stl/`, `www.boost.org`, `www.stlport.org` (Effective STL 항목 50)
- 비표준 결합 컨테이너 – hash_set, hash_multiset, hash_map, hash_multimap
  - 현재는 표준이 아니지만, 해쉬 컨테이너에 충분히 대비해 두자. (Effective STL 항목 25)

컨테이너 형 정리 (Effective STL)

- string 대신 사용되는 `vector<char>`
  - 동적으로 할당한 배열보다는 vector와 string이 낫다. (Effective STL 항목 13)
- 표준 결합 컨테이너 대신 사용되는 vector
  - 결합 컨테이너 대신 정렬된 vector를 쓰는 것이 좋을 때가 있다. (Effective STL 항목 23)
- STL에 속하지 않는 표준 컨테이너 – C++ 배열, bitset, valarray, stack, queue, priority_queue
  - 기존의 C API에 vector와 string을 넘기는 방법을 알아두자. (Effective STL 항목 16)
  - `vector<bool>` 보기를 돌같이 하자. (Effective STL 항목 18)

컨테이너 개념

- 개념적이란 의미는 컨테이너가 상속되지 않음을 의미
  - C++ 문법적 지원이 없음
- 모든 컨테이너 클래스들이 만족해야할 요구 사항들의 집합
- 컨테이너는 단일 형의 다른 객체들을 저장하는 객체 (non-intrusive container)
  - 서로 다른 형의 객체들을 저장하는 intrusive container에 대한 글들도 많이 있으나 표준에선 취급하지 않음
- 컨테이너에 들어갈 객체들을 복사 생성자와 복사 대입 연산자가 구현되어 있어야 함
  - 어떻게 복사가 안되는 클래스 객체를 설계할 수 있을가?
- 기본 컨테이너는 원소들이 특별한 순서로 저장되거나, 저장 순서가 변경되지 않는다고 보장하지 않음
  - 개념의 개량(refinement)은 그러한 보장을 추가할 수 있음

기본 컨테이너의 특성

- 1149 쪽의 표 16.5
- 점근적 복잡도 (asymptotic complexity) — 고정 시간 ($O(1)$), 비례 시간 ($O(n)$) (참고: 컴파일 시간)
- a와 b가 컨테이너일 때, == 연산자는 비례 시간이 걸림
- STL 명세나 함수들을 읽을 때는 점근적 복잡도를 이해해야 하며, STL 함수를 작성하거나 사용했을 때에도 점근적 복잡도를 표기해야 함

시퀀스

- deque, list, queue, priority_queue, stack, vector
- 이터레이터가 최소한 전방 이터레이터는 되어야 한다는 요구 사항 추가
- 원소들이 직선 순서로 배치됨
- 시퀀스 요구 사항 (1151 쪽의 표 16.6)
- 선택적 시퀀스 요구 사항 (1152 쪽의 표 16.7) — $O(1)$에서 수행될 때에만 구현되는 것으로 가정함

vector

- `#include <vector>`
- 배열의 클래스 표현
- 크기의 동적인 변경
- 원소들에 대한 임의 접근
- 말미에 추가나 삭제
- 선두에 추가나 삭제 ($O(n)$)
- 뒤집을 수 있는 가역성 (reversible) 컨테이너 — rbegin(), rend()

```
for_each(dice.begin(), dice.end(), Show); cout << endl;
for_each(dice.rbegin(), dice.rend(), Show); cout << endl;
```

deque

- `#include <deque>`
- 양쪽에 끝이 있는 큐 (double-ended queue) — '데크'라 발음
- vector와 많이 비슷하며, 임의 접근도 지원
- 선두에 추가나 삭제가 $O(1)$
- 대부분의 연산이 선두나 말미의 삽입,삭제라면 고려해 볼만한 구조
- vector보다 더 많은 지원으로 구현은 더 복잡함 — 임의 접근과 시퀀스 중간의 삽입과 삭제는 vector가 더 빠름

list

- `#include <list>`
- 이중 링크드 리스트 — 전후방으로 훑고 지나갈 수 있음
- 어느 위치에서의 삽입도 $O(1)$
- 배열 표기와 임의 접근은 안됨
- 가역성 컨테이너
- 임의 위치에 삽입,삭제 이후에도 원소들은 제자리에 있음

list

- `#include <list>`
- 리스트 지향적인 멤버 함수들 (1155 쪽 표 16.8)
  - insert() — 오리지널 범위를 복사하여 목적지에 삽입
  - splice() — 오리지널 범위를 목적지에 아예 옮김
    - 이터레이터를 그대로 유지함
    - splice()로 접목시키고 난 후에도 이터레이터는 같은 원소를 가리킴
  - unique() — 같은 값들이 인접해 있으면, 그들을 하나의 값으로 단일화함
    - sort()와 unique()을 통해서 여러 동일한 값들을 하나로 단일화할 수 있음

list

- 멤버가 아닌 sort() 함수
  - 정렬을 위해 임의 접근 이터레이터를 요구함
  - 이 함수는 list와 사용할 수 없으므로, list는 자체적인 sort 멤버 함수를 가지고 있음

list 도구 상자

- 두 개의 리스트 합하기 — sort(), merge(), unique()
- sort(), merge(), unique(), remove() 들은 조건 함수 (predicate function) 역할을 하는 함수를 전달인자로 받을 수 있음 — 이 조건 함수는 Java 에서는 Comparator 역할

queue

- `#include <queue>`
- deque 위의 어댑터 클래스
- 큐를 정의하는 기본 연산만 가능 (1158 쪽의 표 16.9)
- 임의 접근을 허용하지 않고, 큐를 훑고 지나가는 이터레이터 연산도 허용하지 않음

priority_queue

- #include <queue>
- vector 위의 어댑터 클래스
- 우선 순위가 더 높은 항목이 큐의 선두로 나감
- 생략 가능한 생성자 전달 인자를 통해 우선 순위를 지정

```
priority_queue<int> pq1;
priority_queue<int> pq2(greater<int>); // greater<int>를 우선 배치함
```

- greater<int> 는 미리 정의된 함수 객체

stack

- `#include <stack>`
- vector 위의 어댑터 클래스
- 전형적인 스택 인터페이스를 제공 (1159 쪽의 표 16.10)
- 임의 접근을 허용하지 않고, 스택을 훑고 지나가는 이터레이터 연산도 허용하지 않음

결합 컨테이너

- 순번 기반이 아닌, 내용 기반 검색
  - associative memory, content addressable memory
  - 해쉉 또는 인덱싱으로 구현
  - 원소들에 대한 빠른 접근을 제공
- 컨테이너 X의 `X::value_type` 은 컨테이너에 저장된 값의 데이터 형
- 결합 컨테이너 X의 `X::key_type` 은 컨테이너에서 사용되는 키의 데이터 형
- `set`, `multiset` (`<set>`), `map`, `multimap` (`<map>`)

set 예제

- 가장 단순한 결합 컨테이너로, 값 자체가 키임
- 키는 고유하여, 하나의 set 에 서로 같은 키는 하나 밖에 없음. (예: {1,2,3,5,7})
- multiset 의 경우, 동일한 키가 여러 개 있을 수 있음. (예: {1,2,2,2,3,5,7,7})
- 가역성임 — 뒤부터 거꾸로 읽을 수 있음

```
set<string> A; // string 객체들의 집합
set<string, less<string> > A; // less<> 템플릿
```

- 다른 컨테이너들처럼 이터레이터의 범위를 전달 인자로 사용하는 생성자를 가지고 있으므로, 집합을 쉽게 배열 내용으로 초기화할 수 있음
- 교집합 (set_intersection()), 합집합 (set_union()), 차집합 (set_difference())

set_union()

- set_union() 함수는 다섯 개의 이터레이터를 사용함 (처음 두 개는 제 1 집합의 범위, 다음 두 개는 제 2 집합의 범위, 마지막 하나는 출력 이터레이터)
- 집합 A와 집합 B의 합집합 출력은?

```
set_union(A.begin(), A.end(), B.begin(), B.end(),
  ostream_iterator<string, char> out(cout," "));
```

- 결과를 집합 C에 넣고 싶다면?
  - C.begin()이 그럴싸해 보이지만, 결합 집합들은 키를 상수값으로 간주하므로, C.begin은 상수 이터레이터이므로 적절하지 않음
  - copy()의 경우처럼, set_union() 함수는 기존의 데이터 위에 덮어쓰며, 새로운 정보를 저장할 충분한 공간이 미리 확보되어야 하는 데, C가 비어있으면 안됨

set_union(), lower_bound(), upper_bound()

- 이 두 문제를 insert_iterator는 같이 해결함 — 출력 이터레이터의 개념을 모델링하는 동시에, 복사를 삽입으로 변환

```
set_union(A.begin(), A.end(), B.begin(), B.end(),
  insert_iterator<set<string> > (C, C.begin()));
```

- lower_bound() — 하나의 키 데이터 형을 전달 인자로 받아서, 그 집합에서 키 전달 인자보다 작지 않은 (즉 크거나 같은) 것 중 가장 작은 (첫번째) 멤버를 지시하는 이터레이터를 반환 — 없으면 end()
- upper_bound() — 하나의 키 데이터 형을 전달 인자로 받아서, 그 집합에서 키 전달 인자보다 큰 것 중 가장 작은 (첫번째) 멤버를 지시하는 이터레이터를 반환 — 없으면 end()

insert()

- set은 insert() 멤버 함수를 가짐

```
string s("tennis");
A.insert(s); // 값을 삽입
B.insert(A.begin(),A.end()); // 범위를 삽입
```

multimap 예제

- 가역성(reversible), 결합성(associative), 소팅됨(sorted)
- `multimap<int,string> codes;` — int를 키 데이터 형으로 사용하고 string을 저장되는 데이터 형으로 사용하는 multimap 객체를 생성
  - 제 3 전달 인자로 비교 함수나 함수 객체를 사용할 수 있으며, 디폴트는 `less<>` 템플릿
- 정보를 한 데 모으기 위해 데이터를 한 쌍으로 결합 — `pair<class T, class U>` 템플릿 사용

```
multimap<int,string> codes;
pair<const int, string> item(51,"Busan");
codes.insert(item);
codes.insert(pair<const int, string> (42,"Daejeon")););
cout << item.first << " " << item.second << endl;
```

multimap 객체에 대한 연산

- count() 멤버 함수로 키들의 개수 반환
- lower_bound() 와 upper_bound() 멤버 함수는 키를 취득하고 set 에 대한 것과 동일하게 동작
- equal_range() 멤버 함수는 키를 전달 인자로 취하여 그 키에 부합되는 범위를 pair로 감싸서 나타내는 이터레이터를 반환
- 1166쪽

```
pair<multimap<int,string>::iterator,multimap<int,string>::iterator>
  range=code.equal_range(51);
cout<<"지역 코드가 51인 도시들:\n";
multimap<int,string>::iterator it;
for(it=range.first;it!=range.second;++it) cout<<(*it).second<<endl;
```

펑크터 개념

- 예

```
for_each(book.begin(),book.end(),ShowReview);

template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f);
```

- 함수 객체 — 1169쪽 윗부분 코드 (operator()(...) 멤버 함수가 오버로딩됨)
- 함수 포인터와 같이 불리기도 함 — 함수 이름, 함수를 지시하는 포인터 변수 (참고: (*(void(*)())0)();)

펑크터 개념

- STL의 펑크터 개념
  - 제너레이터 (generator) — 전달 인자 없이 호출하는 함수
  - 단항 함수 (unary function) — 하나의 전달 인자로 호출하는 함수
    - 프레디키트, 또는 조건 (predicate) — 하나의 전달 인자를 가지고 bool 값을 반환하는 함수
  - 이항 함수 (binary function) — 두 개의 전달 인자로 호출하는 함수
    - 이항 조건 (binary predicate) — 두 개의 전달 인자를 가지고 bool 값을 반환하는 함수

함수 포인터와 펑크터

- list 템플릿은 remove_if() 단항 함수를 멤버 함수로 가짐

```
bool tooBig(int n) { return n>100; }
list<int> scores;
scores.remove_if(tooBig);
```

- 펑크터를 통해 tooBig 에 전달 인자를 하나 더 추가

```
template <class T>
class TooBig
{
  private: T cutoff;
  prublic:
    TooBig(const T& t) : cutoff(t) {}
    bool operator()(const T& v) { return v > cutoff; }
}
```

펑크터를 통한 어댑터 구현

- tooBig이 두 개의 전달 인자를 사용하는 템플릿 함수라면

```
template <class T> bool tooBig(const T& val, const T& lim)
  { return val > lim; }
```

- 이를 하나의 전달 인자를 사용하는 펑크터로 고칠 수 있음

```
template <class T> class TooBig2 {
  private: T cutoff;
  prublic:
    TooBig2(const T& t) : cutoff(t) {}
    bool operator()(const T& v) { return tooBig<T>(v, cutoff); }
}

TooBig2<int> tB100(100);
int x = 0;
cin >> x;
if (tB100(x)) { ... }// if (tooBig(x,100)) 과 동일
```

네 개의 전달 인자를 가지는 transform()

- 처음 두 전달 인자는 입력 컨테이너 안의 범위를 지정하는 이터레이터
- 세 번째 전달 인자는 출력 컨테이너에 결과를 복사할 시작 위치를 지정하는 이터레이터
- 네 번째 전달 인자는 각 원소에 적용할 펑크터 (한 개의 전달 인자)

```
const int LIM=5;
double arr1[LIM] = { 36, 39, 42, 45, 48 };
vector<double> gr8(arr1, arr1+LIM);
ostream_iterator<double, char> out(cout, " ");
transform(gr8.begin(), gr8.end(), out, sqrt);
```

다섯 개의 전달 인자를 가지는 transform()

- 세 번째 전달 인자는 제 2 입력 컨테이너 안의 시작 위치를 지정하는 이터레이터
- 네 번째 전달 인자는 출력 컨테이너에 결과를 복사할 시작 위치를 지정하는 이터레이터
- 다섯 번째 전달 인자는 각 원소에 적용할 펑크터 (두 개의 전달 인자)

```
transform(gr8.begin(), gr8.end(),m8.begin(), out, mean);
```

미리 정의된 펑크터

- 두 배열을 더하고 싶다면...

```
double add(double x, double y) { return x+y; }
transform(gr8.begin(), gr8.end(),m8.begin(), out, add);
```

- 이미 STL이 가지고 있음 (1175 쪽의 표 16.11)

```
#include <functional>
...
plus<double> add; // plus<double> 객체 생성
double y = add(2.2, 3.4); // plus<double>::operator()() 사용
transform(gr8.begin(), gr8.end(),m8.begin(), out, plus<double>());
```

- 해볼만한 숙제 또는 시험 문제
  - transform 만으로 Fibonacci 급수 100 항 구하기
  - transform 만으로 $\sum_{N=1}^{100} N$ 구하기
  - transform 만으로 $x^2 (1 \leq x \leq 100)$ 수열 100 항 구하기

어댑터블 (순응성) 펑크터

- 순응성 제너레이터, 순응성 단항 함수, 순응성 이항 함수, 순응성 조건, 순응성 이항 조건
- 순응성 펑크터들은 typedef 멤버들을 통해 전달 인자와 리턴형을 식별할 수 있음
  - result_type, first_argument_type, second_argument_type
- 벡터 gr8 의 각 원소들에 2.5를 곱한다면?

```
transform(gr8.begin(), gr8.end(),out, ????);
```

- multiplies() 펑크터는 곱셈을 하지만 이항 함수임 — 두 개의 전달 인자를 하나로...
  - 이미 말한 방법 말고도, STL은 bind1st와 bind2nd로 이 문제를 자동화함

```
transform(gr8.begin(),gr8.end(),out,bind1st(multiplies<double>(),2.5));
```

어댑터블 (순응성) 펑크터

- bind1st 는 제 1 전달 인자에 상수 대입, bind2nd는 제 2 전달 인자에 상수 대입
- 순응성 이항 함수 f2()에 제 1 전달 인자로 val 이라는 값을 결합하는 bind1st 객체 f1 은 다음과 같음

```
bind1st(f2,val) f1;
```

알고리즘

- 컨테이너와 사용할 수 있는 STL 비멤버 함수
  - `sort()`, `copy()`, `find()`, `for_each()`, `random_shuffle()`, `set_union()`, `set_intersection()`, `set_difference()`, `transform()`
- 알고리즘 설계의 두가지 일반화 성분
  - 일반형을 제공하기 위해 템플릿 사용
  - 컨테이너에 든 데이터 형에 접근하기 위한 일반화 표현으로 이터레이터 사용
- 일관된 컨테이너 설계 때문에 예를 들어...
  - 객체들의 값 복사 — `copy()`
  - 내용 비교를 위해 — `operator==()`

STL 알고리즘 라이브러리의 분류

- 변경 불가 시퀀스 연산 (algorithm 헤더)
  - 어떤 범위에 들어있는 각각의 원소에 작용하며 컨테이너의 내용을 변경할 수 없음 (find(), for_each())
- 변경 가능 시퀀스 연산 (algorithm 헤더)
  - 어떤 범위에 들어있는 각각의 원소에 작용하며 컨테이너의 내용을 변경할 수 있음
  - transform(), random_shuffle()
- 소팅 및 그와 관련된 연산 (algorithm 헤더)
  - 소팅 함수와 집합 연산을 포함한 여러 가지 함수 (sort())
- 일반화된 수치 연산 (numeric 헤더)
  - 어떤 범위에 있는 내용들의 합계, 두 컨테이너의 내적(inner product), 부분합(partial sum), 인접 차(adjacent difference) 등을 계산
  - vector 가 이러한 연산들에 적합한 컨테이너

알고리즘의 일반적인 특성

- 이터레이터와 이터레이터 범위 사용

```
template<class InputInterator, class OutputIterator>
OutputIterator copy(InputInterator first, InputInterator last,
  OutputIterator result);
```

  - C++ 문법적으로는 그냥 쉽게 T나 U로 표현할 수 있으나, STL 관련 문서화에서는 템플릿 매개 변수의 이름을 통해 그 매개 변수가 모델링하는 개념을 나타냄
  - 선언과 매개 변수들의 이름을 통해 입력 이터레이터, 출력 이터레이터에 대해 문서화

알고리즘의 결과가 놓이는 위치에 따른 분류

- 제자리에 작용하는 알고리즘 (in-place algorithm) — 예를 들어 `sort()`
- 복사본을 생성하는 알고리즘 (copying algorithm) — 예를 들어 `copy()`
- `transform()` 은 둘 다 가능
- 어떤 알고리즘은 제자리 버전과 복사 버전 둘 다 가능
  - 제자리 버전 — `replace()`, `reverse()`, `remove()`, `unique()`, `rotate()`, `partial_sort()`
  - 복사 버전 — `replace_copy()`, `reverse_copy()`, `unique_copy()`, `rotate_copy()`, `partial_sort_copy()`

replace()의 제자리 버전과 복사 버전

- 제자리 버전

```
template<class ForwardInterator, class T>
void replace(ForwardInterator first, ForwardInterator last,
  const T& old_value, const T& new_value);
```

- 복사 버전

```
template<class InputInterator, class OutputInterator, class T>
OutputInterator replace_copy(InputInterator first,
  InputInterator last, OutputInterator result,
  const T& old_value, const T& new_value);
```

  - 복사 알고리즘의 관행은 마지막으로 복사된 값 바로 다음 위치를 지시하는 이터레이터를 반환

조건부로 동작을 수행하는 버전

- `replace_if()`

```
template<class ForwardInterator, class Predicate, class T>
void replace_if(ForwardInterator first, ForwardInterator last,
  Predicate pred, const T& new_value);
```

  - 조건(predicate)은 bool 값을 반환하는 단항 함수
  - 템플릿 매개 변수 이름으로 STL의 개념임을 표현 (Generator, BinaryPredicate 등)

- `copy_if()`는 STL에 없음 (Effective STL 항목 36)

copy_if() 구현 예 (Effective STL 항목 36)

```
template<typename InputInterator, typename OutputInterator,
  typename Predicate>
OutputInterator copy_if(InputInterator begin, InputInterator end,
  OutputInterator destBegin, Predicate p)
{
  while (begin != end)
  {
    if (p(*begin)) *destBegin++ = *begin;
    ++begin;
  }
  return destBegin;
}
```

STL 과 string 클래스

- STL을 염두에 두고 설계되었음
  - `begin()`, `end()`, `rbegin()`, `rend()`
- `sort()` 한 후에 `next_permutation()` (1182 쪽 `strgstl.cpp`)

함수와 컨테이너 메쏘드

- STL 메쏘드와 STL 함수 중 어떤 것을 사용해야 하는가? — STL 메쏘드가 더 낫다.
  - 특별히 그 컨테이너를 위해 최적화되어 있음
  - 템플릿 클래스의 메모리 관리 기능을 이용할 수 있으며, 필요하면 컨테이너의 크기를 조절할 수 있음
- remove 멤버 함수와 메쏘드 (1184 쪽의 `listrmv.cpp`)
  - `l.remove(4);` — 리스트에서 값 4를 모두 삭제하고, 크기는 자동으로 조절
  - `remove(l.begin(), l.end(), 4);` — 값 4가 삭제된 결과 바로 다음으로 이터레이터가 위치하지만, 크기는 그대로임
    - Effective STL 항목 32 — 요소를 정말로 제거하고자 한다면, remove 류의 알고리즘에는 꼭 erase를 붙여서 사용하자.
    - `l.erase(remove(l.begin(), l.end(), 4), l.end());`

STL 사용하기

- 단어들을 입력받아 빈도를 저장하는 프로그램 (1188 쪽의 usealgo.cpp)
  - push_back(), sort(), unique(), transform(), count()
- map 객체 사용

```
map<string, int> wordmap;
set<string>::iterator si;
for (si=wordset.begin(); si!=wordset.end();si++)
  wordmap.insert(pair<string,int>(*si,count(words.begin(),
    words.end(),*si)));
```

- map 객체 사용 — 더 나은 방법 (wordmap["the"])

```
for (si=wordset.begin(); si!=wordset.end();si++)
  wordmap[*si]=count(words.begin(),words.end(),*si);
```

STL 을 남발하는 쓰기 전용 코드 하나

- vector<int> 에서 x보다 작은 값을 다 지우되, y보다 크거나 같은 값 중 가장 마지막에 오는 것 앞에 있는 것들은 모두 그대로 두고 싶다.

STL 을 남발하는 쓰기 전용 코드 하나

- vector<int> 에서 x보다 작은 값을 다 지우되, y보다 크거나 같은 값 중 가장 마지막에 오는 것 앞에 있는 것들은 모두 그대로 두고 싶다.
- ```
  vector<int> v;
  int x,y;
  ...
  v.erase(remove_if(find_if(v.rbegin(),v.rend(),
    bind2nd(greater_equal<int>(),y)).base(),v.end(),
    bind2nd(less<int>(), x)),v.end());
  ```
  ```

STL 을 남발하는 쓰기 전용 코드 하나

- ```
  v.erase(
    remove_if(
      find_if(
        v.rbegin(),v.rend(),bind2nd(greater_equal<int>(),y)
      ).base(),
      v.end(),
      bind2nd(less<int>(), x)
    ),
    v.end()
  );
  ```
  ```

STL 을 남발하는 쓰기 전용 코드 하나

- ```
  v.erase(
    remove_if(
      find_if(
        v.rbegin(),v.rend(),bind2nd(greater_equal<int>(),y)
      ).base(),
      v.end(),
      bind2nd(less<int>(), x)
    ),
    v.end()
  );
  ```
- Effective STL 항목 47
- 여기서 쓰기 전용 코드란 프로그래머가 코드를 짤 때는 쉬운 데, 남이 읽기 어려운 코드를 말함
- 이런 코드는 잘난 척 할 때 말고는 피하자.
  ```

STL 을 남발하는 쓰기 전용 코드 하나

- 프로그램은 사람이 읽을 수 있도록 쓰여져야 하고, 컴퓨터가 실행할 수 있도록 하는 것은 그 후의 문제이다 — 해롤드 아벨슨 (Harold Abelson), 제랄드 서스맨 (Gerald Sussman)
- 프로그램을 쓸 때는 사람을 먼저 생각하고, 컴퓨터는 두번째로 생각하라 — 스티브 맥코넬 (Steve McConnell)
- 대안

```
typedef vector<int>::iterator VecIntIter;

VecIntIter rangeBegin=find_if(v.rbegin(),v.rend(),
  bind2nd(greater_equal<int>(),y)).base();
v.erase(remove_if(rangeBegin,v.end(),bind2nd(less<int>(),x)),v.end());
```

기타 라이브러리

- complex 헤더 파일은 float, long, long double 을 위한 특수화를 통해 complex 템플릿 클래스 제공

vector 와 valarray

- vector 템플릿 클래스는 컨테이너 클래스와 알고리즘으로 구성된 시스템의 일부
  - 소팅, 삽입, 재배치, 검색, 다른 컨테이너로의 데이터 전송 등의 컨테이너 지향적인 활동을 함
- valarray 템플릿 클래스는 수치 계산을 지향하며 STL의 일부가 아님
  - push_back() 이나 insert() 메쏘드가 없음
  - sum(), size(), max(), min() (1192쪽)

vector 와 valarray

- 다음의 선언들을 가정해 보자

```
vector<double> ved1(10), ved2(10), ved3(10);
valarray<double> vad1(10), vad2(10), vad3(10);
```

- 두 배열의 원소들의 합을 세 번째 배열에 대입
  - vector 클래스의 경우

```
transform(ved1.begin(), ved1.end(), ved2.begin(), ved3.begin(),
  plus<double>());
```

  - valarray 클래스의 경우

```
vad3 = vad1 + vad2;
```

vector 와 valarray

- 한 배열의 원소들에 각각 2.5를 곱하여 대체 ‘

```
transform(ved3.begin(), ved3.end(), ved3.begin(),
  bind1st(multiplies<double>(), 2.5));

vad3 = 2.5 * vad3;
vad3 *= 2.5;
```

- 한 배열의 원소들에 각각 자연 대수를 취하여 두번째 배열에 대입

```
transform(ved1.begin(), ved1.end(), ved3.begin(), log);

vad3 = log(vad1); // 오버로딩된 log()
vad3 = vad1.apply(log); // 오버로딩되지 않은 log()
```

vector 와 valarray

- 시험 문제 — 다음의 vector-STL 버전은?

```
vad3 = 10.0 * ( (vad1 + vad2) / 2.0 + vad1 * cos (vad2) );
```

- valarray 클래스는 수학 연산에 대한 명쾌한 표기상의 이점이 있으나, 융통성이 부족함
  - resize() 가 있으나 push_back() 은 없으며, 삽입, 검색, 소팅을 위한 메쏘드가 없음
- 1195쪽 valvect.cpp

STL과 valarray를 같이 사용할 수 있을까?

- 10 개의 원소를 가지는 valarray 객체 — `valarray<double> vad(10);`
- `begin()` 과 `end()`는 없음 — `sort(vad.begin(), vad.end())` 안됨
- `vad` 는 주소가 아니라 객체임 — `sort(vad, vad+10)` 안됨
- 주소 연산자는 일단 되는 거처럼 보임 — `sort(&vad[0], &vad[10])`
  - 배열 크기와 같거나 더 큰 인덱스를 사용할 경우, 정의되지 않은 행동을 초래함 — 예를 들면 `sort()`가 안될 수도 있음
  - 처음에 valarray 객체의 크기를 필요한 원소보다 하나 더 많게 하면, `sum()`, `max()`, `min()`, `size()` 에서 문제를 일으킬 수 있음

valarray 의 그 외의 기능

- 10 개의 원소를 가지는 valarray 객체 — `valarray<double> numbers(10);`
- 각각의 원소가 9보다 크면 true — `valarray<bool> vbool = numbers > 9;`
- `slice()` 는 인덱스를 지정하여 부분 집합을 만드는 기능
  - `slice(1,4,3)` — 1, 4, 7, 10
  - `varint` 가 `valarray<int>` 객체라면 `varint[slice(1,4,3)]=10` 은 1,4,7,10 번째 원소들을 10으로 설정
    - `slice(0,3,1)` — 0,1,2
    - `slice(0,4,3)` — 0,3,6,9