1.1 기본 아이디어

STL은 개념적으로 크게 네 가지 라이브러리로 나눌 수 있다.

* 컨테이너 라이브러리에는 데이터를 저장하고 관리하는 데 쓸 수 있는 컨테이너가 정의되어 있다. 종류 : array, vector, stack, queue, deque, list, forward\_list, set, unordered\_set, map, unordered\_map이 있다.
* 반복자 라이브러리에는 반복자가 정의되어 있으며 반복자는 포인터처럼 동작하는 객체로 컨테이너에 있는 객체의 순차열을 참조할 때 사용된다. iterator 헤더 파일 하나로 정의됨.
* 알고리즘 라이브러리에는 컨테이너에 저장된 원소에 적용할 수 있는 알고리즘이 폭넓게 정의되어 있다. 알고리즘 헤더파일은 algorithm헤더 파일이다.
* 수치 라이브러리에는 숫자와 관련된 기능이 폭넓게 정의되어 있으며 여기에는 컨테이너에 있는 원소에 수치 처리를 할 수 있는 기능이 포함되어 있다. 헤더 파일로는 complex, cmath, valarray, numeric, random, ratio, cfenv가 있다.

1.2 템플릿

템플릿은 함수나 클래스를 매개변수로 작성한 명세.

예) template <typename T> T& larger(T& a, T& b){ return a > b ? a : b;}

클래스 템플릿

template <typename T> class Array {};

Array& operator=(const Array& rhs)=delete; // 할당 연산자 없음

일반적으로 복제 생성자, 이동 생성자, 복사 할당 연산자, 이동 할당 연산자, 소멸자 중에 하나라도 정의해야 한다면 클래스 멤버 다섯 개를 모두 정의해야 하며, 정의 하고 싶지 않는 클래스 멤버는 delete로 지정해야 한다.

멤버 함수가 타입 매개변수 T를 전혀 사용하지 않아도 같은 헤더 파일에 둔다. size()를 클래스 템플릿 안에 정의하지 않아도 size()는 템플릿으로 정의되어야 한다. 멤버 함수를 정의하는 템플릿에서 사용하는 타입 매개변수는 클래스 템플릿에서 사용한 타입 매개변수와 반드시 일치해야 한다.

template<typename T> // 매개변수 T로 정의한 함수 템플릿

Array<T>::Array(size\_t arraySize) try : elements {new T[arraySize]}, count {arraySize}{}

catch(const std::exception& e)

{ st::cerr << “Array 생성자에서 메모리 할당 실패.” << std::endl; rethrow e; }

elements에 메모리를 할당하다가 예외가 발생할 수 있으므로 생성자는 try 블록 함수여야 한다.

템플릿 타입 매개변수는 함수 템플릿 정의가 클래스 템플릿에 묶여 있으므로 템플릿 타입 매개변수는 생성자 이름을 한정할 때 반드시 쓰인다. typename은 템플릿 매개변수 목록에만 쓰여야 한다.

template<typename T>

inline Array<T>::Array(const Array& other)

try : elements {new T[array.count]}, count {array.count}

{

for(size\_t I {}; I < count; ++i){ elements[i] = array.elements[i]; }

}catch (std::bad\_alloc&)

{

std::cerr << “Array 객체 복제를 위한 메모리 할당에 실패했습니다.” << std:: endl;  
}

이 코드는 할당 연산자가 타입 T에 대해 동작한다고 가정하고 있다.

컴파일러는 클래스 템플릿을 템플릿이 생성한 타입을 갖는 객체로 인스턴스화한다. 다음 코드를 보자.

Array<int> data {40}; 이 코드가 컴파일될 때 세 가지 처리가 일어난다. 먼저 Array<int>클래스 정의가 생성, 타입 식별, 생성자 정의, 소멸자 정의 이다. data 객체의 정의만 본다면 클래스는 다음과 같이 정의된다.

class Array<int>

{

private:

int\* elements;

size\_t count;

public:

explicit Array(size\_t arraySize);

virtual ~Array();

};

여기서 볼 수 있는 멤버 함수는 생성자와 소멸자뿐이다. 컴파일러는 객체 생성에 필요 없는 나머지 부분은 인스턴스로 생성하지 않는다.

타입 별칭도 템플릿으로 정의할 수 있다. 예)

template<typename T> using ptr = std::shared\_ptr<T>;

여기서 템플릿은 스마트 포인터 템플릿 타입 std::shared\_ptr<T>에 대한 별칭으로 ptr<T>를 정의 한다. 이렇게 하면 std::shared\_ptr<std::string> 대신 ptr<std::string>으로 사용할 수 있다.

using std::string; 을 쓰면 ptr<string>으로 사용할 수 있다.

1.3 컨테이너

순차 컨테이너는 객체들을 선형으로 저장한다. 선형은 배열과 비슷하지만, 반드시 연속 메모리로 저장될 필요는 없다는 뜻이다. 멤버 함수를 호출하거나 반복자를 사용해서 객체들을 차례로 접근할 수 있다. 또한, 때에 따라서는 첨자 연산자와 인덱스를 사용해 접근할 수도 있다.

연관 컨테이너는 객체들을 연관된 키와 함께 저장한다. 키를 이용해서 연관 컨테이너에서 객체를 가져온다. 반복자를 사용해서도 연관 컨테이너에서 객체들을 가져올 수 있다.

컨테이너 어댑터는 순차 컨테이너나 연관 컨테이너에 저장된 데이터에 접근하는 다른 방법을 제공하는 어댑터 클래스 템플릿을 말한다.

모든 STL 컨테이너는 객체를 컨테이너에 저장할 때 복제본을 저장한다. 이동 생성자와 이동 할당 연산자가 반드시 noexcept로 지정되어야 한다.

컨테이너는 객체들을 힙에 저장하며 객체가 차지하는 공간도 자동으로 관리한다. 타입 T의 객체들을 저장할 컨테이너에 공간을 할당하는 작업은 할당자가 담당하며, std::allocator<T> T의 객체를 힙 메모리에 할당하는 할당자라는 뜻. 할당자도 직접 정의해서 쓸 수 있다.

타입 T객체를 컨테이너에 저장하려면 최소한 다음과 같은 모습이어야 한다.

class T

{

public:

T(); // 기본 생성자

T(const T& t); // 복제 생성자

~T(); // 소멸자

T& operator=(const T& t); // 할당 연산자

};

operator<()는 T의 정의에 포함되지 않았다. 따라서 operator<()를 정의하지 않은 타입의 객체는 map이나 set같은 연관 컨테이너에서 키로 사용할 수 없으며 보다 작은(<) 연산을 지원하지 않는 원소들을 담은 순차 컨테이너에는 sort()나 merge()같은 정렬 알고리즘을 적용할 수 없다.

1.4 반복자

반복자는 컨테이너의 종류와 상관없이 컨테이너에 있는 원소들과 알고리즘을 이어준다. end반복자는 순차열에서 마지막 원소가 아니라 마지막 원소에서 하나 더 지나친 위치를 가리킨다.

배열, 컨테이너 또는 문자열 객체의 const 반복자를 반환하는 전역 함수 cbegin()과 cend()도 있다. const 반복자는 변경되지 말아야 할 것을 가리키지만, 반복자 자체는 바꿀 수 있다.

1.4.2 반복자 카테고리

모든 반복자 타입은 복사 생성자, 복사 할당 연산자(operator=()), 소멸자를 가져야한다. 반복자가 가리키는 개체들은 교환 할 수 있어야한다.

반복자 카테고리

* 읽기 반복자는 객체에 대한 읽기 권한을 갖는다. Iter가 입력 반복자라면 iter가 가리키는 값에 대한 참조를 얻는 \*iter를 반드시 지원해야 한다. 입력 반복자는 한 번만 읽을 수 있다. 즉, 한번 반복자를 증가시키면 반복자가 가리켰던 이전 원소에 접근 할 수 없으며 이전 원소에 접근하려면 새 반복자를 써야 한다. 순차열을 읽고 싶을 때마다 새 반복자를 생성해야 한다. 입력 반복자에서 할 수 있는 연산은 다음과 같다.

++iter, iter++, iter1 == iter2, iter1 != iter2, \*iter 로 감소 연산자가 없다. 입력 반복자는 iter->member 같은 표현식을 쓸 수 있다.

* 출력 반복자는 객체에 대한 쓰기 권한을 갖는다. Iter가 출력 반복자라면 새로운 값을 할당할 수 있으므로 \*iter = new\_value라고 쓸 수 있다. 출력 반복자는 한 번만 쓸 수 있다. 즉, 순차열에 쓰기를 하고 싶을 때마다 새 반복자를 생성해야 한다. 출력 반복자에서 할 수 있는 연산은 다음과 같다.

++iter, iter++, \*iter 감소 연산자가 없는 것에 주의하자. 오직 쓰기 권한만 얻을 수 있다. 따라서 출력 반복자로는 iter->member 같은 표현식을 쓸 수 없다.

* 순방향 반복자는 입력 반복자와 출력 반복자의 기능에 몇 번이고 쓸 수 있는 기능을 더한 것이다. 따라서 원소를 읽거나 쓰는 작업에 순방향 반복자를 몇 번이고 원하는 만큼 재사용할 수 있다. 수행하는 작업에 따라 순방향 반복자가 필요한지 결정된다. 예를 들어 범위를 검색해서 원소들을 대체하는 replace() 알고리즘을 사용하려면 대체할 원소를 가리키는 반복자가 원소를 덮어쓸 때도 재사용될 수 있어야 하므로 순방향 반복자가 필요하다.
* 양방향 반복자는 순방향 반복자와 같은 기능에 역방향으로 이동할 수 있다. 따라서 반복자를 증가시켜 다음 원소로 이동하는 것도 가능하고 --iter나 iter--같은 감소 연산자로 이전 원소로 이동하는 것도 가능하다.
* 랜덤 액세스 반복자는 양방향 반복자와 같은 기능에 원소들에 마음대로 접근할 수 있는 기능을 더한 것이다.

정수만큼 증가하거나 감소: iter + n, iter – n, iter += n, iter -= n

정수로 인덱스 접근: iter[n], \*(iter + n)으로 표현한 것과 같음

두 반복자의 차이 : iter1 - iter2, 원소들의 차이를 정수로 반환

반복자 비교: < , > , <= , >=

범위로 지정된 원소들을 정렬하려면 범위를 랜덤 액세스 반복자로 지정해야 한다. 랜덤 액세스 반복자는 첨자 연산자 [ ]를 사용할 수 있다. 반복자 first가 있다면 first[3]은 \*(first + 3)과 같으며 4번째 원소에 접근하라는 뜻이 된다. 반복자의 첨자 연산자에 사용하는 인덱스는 검사되지 않는다. 인덱스 값이 적법한 범위를 벗어난 값이어도 이를 막을 방법이 없다.

각 반복자 카테고리는 iterator 템플릿에 타입 인수로 사용된 빈 클래스, 즉 반복자 태그 클래스로 확인할 수 있다. 반복자 태그 클래스의 유일한 목적은 특정 반복자 타입이 무엇을 할 수 있는지 지정하고 iterator 템플릿 타입 인수로 사용하는 것이다.

컨테이너가 갖는 반복자의 특징은 컨테이너 타입에 따라 결정된다. 예를 들어 vector와 deque컨테이너는 랜덤 액세스 반복자를 갖는다. 이들 컨테이너는 원소들에 랜덤하게 접근할 수 있어야 하기 때문이다. 반면에 list와 map 컨테이너는 항상 양방향 반복자를 제공한다. 이들 컨테이너는 원소들에 랜덤하게 접근할 필요가 없기 때문이다. 알고리즘에는 최소 기능 수준만 반영하므로 알고리즘 타입에는 입력, 출력, 순방향 반복자 타입이 일반적으로 사용된다.

배열에서 반복자 사용 예제

#include<numeric> // accumulate()함수 – 범위에 속한 원소들의 합게

#include<iostream> // 표준 스트림

#include<iterator> // 반복자, begin(), end() 함수

int main()

{

double data[ ]{2.5, 4.5, 6.5, 5.5, 8.5};

std::cout << “배열 원소들:\n”;

for(auto iter = std::begin(data); iter != std::end(data); ++iter)

std::cout << \*iter << “ “;

auto total = std::accumulate(std::begin(data), std::end(data), 0.0);

std::cout << “\n배열 원소들의 합계: “ << total << std::endl;

}

accumulate()함수는 덧셈을 지원하는 타입이어야 동작하며 operator+()를 정의한 클래스 타입의 객체들과도 동작한다. 또한 기본 연산 +이 외에 다는 이항 연산을 지정할 수도 있다.

1.6 스마트 포인터

스마트 포인터는 자유 공간에 할당된 메모리의 주소만 저장할 수 있다.(new를 사용한 공간)

원시 포인터에서 하던 증가, 감소 같은 산술 연산은 스마트 포인터에서 할 수 없다.

unique\_ptr<T>는 타입 T에 대한 포인터처럼 행동하며 유일해야 한다. 유일하다는 것은 둘 이상의 unique\_ptr객체가 같은 주소를 소유할 수 없다. 즉, 다른 객체에게 할당하거나 복제도 할 수 없다는 것.

shared\_ptr<T> 는 같은 주소를 담은 shared\_ptr<T>객체가 얼마든지 있을 수 있다는 것. 주소가 같은 객체의 개수는 기록된다. (카운트 증감) 카운트가 0이 되면 자동 해제됨. 가리킨 객체도 소멸됨.(자동 해제)

weak\_ptr<T>는 shared\_ptr<T> 객체에서 생성해서 연결하고 같은 주소를 가리킨다. 그런데 shared\_ptr<T>카운트가 증가하지는 않는다. shared\_ptr<T> 참조가 소멸하거나 다른 주소로 재할당될 때 shared\_ptr<T>의 메모리는 해제되더라도 연관된 weak\_ptr<T>객체는 남아 있게 된다. weak\_ptr을 쓰는 이유는 shared\_ptr객체로 부주의하게 순환 참조(양방향참조)를 만들기 때문이다. 이렇게 되면 어떤 객체도 소멸할 수 없다.(카운트가 0이 될수 없다.) shared\_ptr객체가 가리키는 객체를 weak\_ptr객체를 사용해 가리키면 순환 참조를 피할 수 있다.

unique\_ptr<T> 포인터 사용하기

unique\_ptr<T>가 소유한 객체는 원시 포인터를 이용해서 객체에 대한 접근을 허용할 수 있다.

unique\_ptr 생성법

std::unique\_ptr<std::string> pname {new std::string {“Algernon”}};

힙에 생성된 string 객체가 unique\_ptr<string> 생성자에 전달된다. 기본 생성자는 unique\_ptr<T>객체의 내부 원시 포인터를 nullptr로 생성한다.

auto pname = std::make\_unique<std::string>(“Algernon”); // memory헤더에 정의된 make\_unique<T>()함수 템플릿을 사용하는 것.

unique\_ptr은 복제할 수 없으므로 복사생성자가 호출되는데는 쓸 수 없다. 즉, 함수 파라미터로 값이 들어있어서는 안되고 &가 붙어야 함수에 인자로 쓸 수 있다.

원시포인터를 얻는 방법

자료형 pstr {unique\_p.get()}; // 멤버함수 get()은 unique\_ptr<T>에 있는 원시 포인터를 반환한다.

p45.

스마트 포인터가 소멸될 때 포인터가 가리키는 객체도 소멸된다. reset()을 호출하면 유니크 포인터가 가리키는 객체를 소멸시키고 포인터는 nullptr이 된다.

p46.

auto pname = make\_unique<string>(“Algernon”);

pname.reset();

reset의 매개변수에 새 객체를 전달하면 이전에 가리키던 객체는 소멸되고 새 객체를 가리킨다. reset(new string(“Fred”));

스마트 포인터에 release()를 호출하면 unique\_ptr객체가 가리키는 객체를 반환하고 자신(unique\_ptr)은 nullptr이 된다.

swap을 쓰면 포인터가 소유한 객체를 상호 교환도 할 수 있다.

포인터.swap(교환대상포인터);

p47

unique\_ptr객체를 비교 검사하기

unique\_ptr이 어떤 객체를 가리키면 true를 반환한다.

**shared\_ptr 사용하기**

shared\_ptr을 정의할 때 가리키는 객체를 힙에 올리고 복제본 개수를 기록하는 작업을 객체를 힙에 올리기 때문에 총 2개를 힙메모리에 할당하므로 처리 시간이 비싸다.

따라서 make\_shared<T>() 함수를 사용해서 shared\_ptr<T> 타입의 스마트 포인터를 생성 다면 더 효율적으로 할 수 있다.

shared\_ptr는 정의할 때 다른 shared\_ptr로 초기화 할 수 있다.

shared\_ptr<double> s1 {s0}; // s1은 s0와 같은 객체를 가리킨다.

auto s2 = s1; // s2는 s1와 같은 객체를 가리킨다.

get을 이용하여 얻은 원시 포인터를 이용해서 shared\_ptr을 생성하는 건 정의되지 않은 행동이므로 하지 말자

shared\_ptr 초기화 하기

shared\_ptr객체에 nullptr을 할당하거나 reset() 카운트 감소. 하지만 카운트가 0이 아니면 가리키는 객체는 여전히 살아있음.

p50

shared\_ptr객체를 비교검사하기

같은 객체 가리키는 확인하기

if((pA == pB) && (pA != nullptr))

cout << “두 포인터는 같은 객체를 가리킵니다.” << endl;

shared\_ptr객체 복제본 유무 검사

if(pA.unique())

cout << “단 하나” << endl;

unique()함수는 카운트가 1이면 true를 반환 함.

use\_count()는 카운트를 반환함 0이면 nullptr을 가리키는 것.

p51.

**weak\_ptr<T> 포인터**

weak\_ptr포인터는 shared\_ptr로만 생성할 수 있다. weak\_ptr는 주로 클래스 안의 멤버 변수를 초기화할 때 이용한다. 왜 shared\_ptr을 쓰지 않냐면 상호참조로 인해 객체를 접근도 못하고 지울 수도없는 상황을 피하기 위해서 이다. weak\_ptr은 배열에 있는 외부 포인터들이 소멸되거나 초기화 되었을 때 객체들이 소멸하는 것을 방해하지 않는다.

weak\_ptr<T> 생성 방법

auto pData = make\_shared<T>(); // 일단 weak\_ptr을 만들기 위해 shared\_ptr을 만듬.

weak\_ptr<T> pwData{pData}; // 공유 포인터에서 약한 포인터 생성.

weak\_ptr<T> pwwData{pwData}; // 약한 포인터에서 약한 포인터 생성.

p52

약한 포인터로는 할 수 있는게 많지 않다. 역참조해서 접근 조차 할 수 없음

약한 포인터는 shared\_ptr이 있는지 확인 가능하고 weak\_ptr객체에서 shared\_ptr객체를 생성할 수 있다.

**테스트 방법**

if(pwData.expired())

cout << “객체가 존재하지 않습니다.” << endl;

pwData객체의 expired()함수는 객체가 없으면 true를 반환한다.

**weak\_ptr으로 shared\_ptr객체 만들기**

shared\_ptr<T> pNew {pwData.lock()};

lock()함수는 pwData가 가리키는 shared\_ptr이 존재하면(expired()가 false를 반환할 때) 새로운 shared\_ptr객체를 반환하면서 객체를 잠근다. 만약 존재하지 않으면 nullptr로 초기화된 shared\_ptr을 반환한다.

p. 53

알고리즘

분류

변경 불가 순차열 연산 – 대상 내용을 변경하지 않는 것. 주어진 값과 일치하는 원소를 찾는 알고리즘이 원본 데이터를 변경할 필요는 없다. 예) find(), count(), search(), equal()

변경 가능 순차열 연산 – 반대로 대상 내용을 변경하는 것. 예) swap(), copy()

순차열 순서 변경 – 예) sort(), merge(), min(), max()