< 고급언어의 사용이유 >

이유 :어셈블리어는 CPU의 명령어와 1대1로 대응되는 언어로 CPU가 바뀌면 기존에 작성한 코드가 작동하지 않는다.

정리하면 : 어셈블리어는 CPU에 의존적이기 때문에 고급언어를 사용한다. 고급언어는 CPU에 독립적이다'라 할수있다.

< C언어의 장점 >

대부분의 CPU는 개발환경으로 C컴파일러를 제공하기 때문에 이식성이좋다.

< 빌드와 컴파일의 다른 점 >

빌드는 크게 컴파일과 링크로 이루어진다.

컴파일 단계에서 소스코드들은 오브젝트파일로 변환되고

링크를 통해해 오브젝트들은 서로 연결된다.

즉 컴파일은 빌드의 한 부분이다.

빌드가 완료되면 실행파일로 만들어진다.

< 2의 보수 Why_ (왜 1을 더하는가) >

데이터를 반전 후 1을 더 하는것이 음수를 표현하는 방법이다.

1111 1111 1111 1111을 낭비하게 된다.

이게 핵심 -0 과 0을 중복으로 표현할 필요가없다.

=> 0과 -0의 존재 때문에 -0없앤 대신 -를 표현하는 수가 1이 더 많다.

< 검증의 효과 >

2의 보수값과 변환전의 값을 더하면 비트범위 값은 0이된다.

Ex) 0000 0000 0000 0001

+ 1111 1111 1111 1111 = 0000

0000 0000 0000 0001

<2의 보수의 단점 overflow>

- **최댓값에 + 1 => 최솟값**
- **최솟값 -1 = 최댓값이된다.**
- **Ex) 2바이트 : 32767 + 1 = -32768**
- **-32768 – 1 = 32767**

Unsigned 의 최댓값 + 1 = 0 이 됨
=> limits.h를 참고

<전역 변수>

Why_ 다른 외부함수에서도 사용할 수 있다.
How_ 전역 변수임을 알리기위해 g_를 붙인 변수명을 사용
How2_ 절차형이기때문에 사용될 외부 함수보다 상단에 위치해야함.
When_ 웬만하면 사용되지않는다 => (독립성을 잃기때문에)
대체 : 추후에 배울 함수의 매개변수로 대체사용된다.

< 함수 >

- **Why_ 변수를 여러곳에서 사용하고 싶어서 사용함**
- **전역변수가 아닌 함수를 사용하게되면 독립성이 올라감**

How_ call by value & Reference
How2_ Swap(int a, int b); Swap(int& a, int& b);
When_ 변수를 외부함수에서 쓰되 변경시키고 싶을때는
Reference를 아닐때는 value를 넘겨준다.

<배열>

Why_ int a1, a2,a3,a4,a5,a7,a8.....an;

=> 이렇게 반복적으로 변수명을 정의하는 상황방지

How_ <T> a[n]; 선언

How_ sizeof(arr) = arr의 크기를 반환함 int 형 4개인경우 16 반환 How_ Index를

이용해서 접근을 할수있는 이유는 선언문의 의미가

T타입의 크기만큼 n개의 메모리 주소를 연속적으로 사용함을 선언한것이다.

Sizeof(arr[0) = 데이터형에 맞는 Byte를 반환함함

When_ 같은 타입의 변수가 여러 개 필요할 때 사용

'메모리에 연속적으로 적용된다'는 것이 포인트

< 문자열 리터럴>

Const char* str = "ABCED" ;같은 것

보통 상수는 임시값이라고 쓰일 떄마다 레지스터에 생성되어 사용되지만.

문자열의 경우 크기가 워낙 변동적이라 레지스터가 아닌 메모리에 저장이 됌

=> 즉 상수는 레지스터에 저장되기떄문에 우리가 변경하는 접근이 불가능하나.

문자열 상수는 메모리에 저장되기 때문에 접근이 가능해짐 이로인해 문제가 발생함.
Str[1] = 'c' 와 같은 변경이 불가능함.

<문자열 변수>

Char str[] = "ABCED" ; 로 초기화가 가능
문자열의 경우 끝에 '₩0' 라는 널문자가 있어야 하는데 자동적으로 추가가 됨
Char str[5] = "AB" => 'A' 'B' '₩0' '₩0' '₩0' 으로 저장장
// 직접 크기를 정의할경우 널문자를 생각하여 공간을 배정해야함.
Str = "cdcs" ; // 컴파일 에러 가 발생함 배열의 이름은 str은 첫 주소를 나타냄
Str[1] = 'c' 와 같은 변경이 가능 함

< string 헤더파일 >

Strcmp(str,"dsa") == 0 이면 같음
Strcpy(str,"asd")로 복사 변경경
Append 기능 strcat(str,"dasd); //이어 붙이기기
Gets(str)// getline(cin,str)같은 기능 // 한줄 입력
Puts(str); // 한줄 출력

< 포인터 >

포인터의 크기는 운영체제에 의해 결정된다. 64bit => 8byte 32 => 4byte

메모리에 직접 접근할수있는 수단임

Why_ 메모리에 기록된 변수 값에 접근하기 위해서 사용 됌

= > 사칙연산 int arr[4];

arr = 첫주소 0x13FF2309 였다고 가정하면

Int *p = arr; // 라고 하였을때

p + 1 은 0x13FF230A 가 될까?

포인터 의 타입을 계산하여 더 해진다.

0x13FF2309 + (n * Sizeof(int))가 됌

이 때 data 타입의 크기는 4/8/12/16/20/24/28/32

4의 배수로 진행됌

Struct 의 메모리 정렬

When_ 배열은 같은 데이터 타입을 연속으로 생성하지만

다른 데이터 타입이 같이 필요한 경우가 있다.

Ex) Student(name, score)

How 구조체 생성 시에 다른 타입의 변수와 합쳐질 때 가장 큰 데이터 타입의 배수로

메모리에 저장되는 현상 *Padding 현상을 가짐

Ex) Struct student {

Double number;

Char name[4]';} // 일경우 8의 배수로 저장 공간이 할당 됨.

모자른 경우 사용하지 않는 데이터를 메모리에 추가해서 맞추는 Padding이라는 작업이 실시 됨.

How2_* Padding이란 cpu가 메모리에 접근을 할때 4/8/16/32 의 배수로 읽는다고 한다.

만약 배수가 안된다면 데이터가 들어있지않는 빈공간의 메모리로 사이즈를 맞추는 현상을 패딩현상이라고 한다.

<typedef>

Why,How_ typedef int Energy;

Damaged(int hp,int damage){ hp -= damage;}

라는 함수가 있다고 생각해보자.

프로젝트를 진행하다가
Energy의 타입을 Float 로 바꾸고 싶다면
Energy를 사용하는 모든 함수의 타입도 Float로 바꿔야 할 것이다.
그 경우 typedef float Energy; 로 바꿔주면
손쉽게 변경되고 변수 명을 잘 지으면 가독성도 오른다는 잠재력이 있다.

< 기억 부류 >

| auto | 변수 앞에 작성하지 않으면 Default로 오게된다.

//블록에 들어가고 나오면 자동 할당 해제

| extern | 함수 앞에 작성하지 않으면 Defalut로 오게된다.

Why_

#include “다른 헤더파일” 를 사용해서 사용하는 함수들은 모두 extern은 // 다른 곳에서 선언을 하였다는 가정이라고 할 수 있다.

외부 소스에서 사용하고 싶은 함수나 변수가 있을 때가 있다.

How 헤더파일에서 선언문을 작성하고 cpp파일에서 함수를 작성

할 수 있는 이유도 함수 앞에 묵시적으로 extern이 정의되어 있기 때문이다.

When_ extern : 외부에서 함수의 바디를 작성할 때

When_ 동일한 클래스 객체가 공용으로 사용하고 싶은

변수나 함수가 존재할 때

Static을 사용하여 오브젝트를 생성하지 않아도 사용 가능하게하기

How _ Static은 메모리에 생성되는 과정이 다른 변수나 함수와는 다르다.

객체가 생성될 때 메모리에 올라가는 다른 타입과 달리

Static은 실행 즉시 메모리에 생성되게 된다.

그 원리로 Static으로 선언한 함수는 딱히 객체를 생성하지않아도

Class명::함수명 을 이용해 사용이 가능하다.

% 주의점 : Static 함수를 사용할때 클래스 내부의 변수는 Static변수만 사용이 가능하다. =>

이유: 객체마다 변수의 메모리가 다르기때문에 특정할 수 없다.

(객체가 생성 안됬을수도있음)즉 Static을 사용하여 메모리를 특정해야한다.

Static과 Static instance변수, 생성자를 연동시켜서

싱글톤 패턴이 만들어지는거임

< Singleton 싱글톤 >

```
Class SingleTon
{
Private:  //생성자와 소멸자를 Private로 구현한다.
인스턴스 구현방법 2가지
Static SingleTon* Instance;// 포인터로 구현 할 것인가?
Static SingleTon instance; //변수로 구현 할 것인가.
```

정적 멤버 변수도 전역변수의 특성을 갖기 때문에

외부에서 초기화를 해줘야 사용할 수 있다.

전역 변수처럼 선언과 동시에 초기화가 안됨

<초기화 방법>

SingleTon* SingleTon::Instance;

SingleTon SingleTon::instance;

Public: // 인스턴스를 가져오기

Static SingleTon* Get_pInstance(){return Instance;}

Static SingleTon& Get_instance() {return instance;}

// 포인터의 경우 nullptr 일때 New singleton() 해서 리턴하기

<의문>

정적 Class 멤버함수로 Instance를 선언하지않고

Get_instance(){static Singleton instance; return instance;}

// Get함수 내부에서 정적 변수를 선언하니 외부에서 초기화 하지않아도 실행되었다. 이유가 무엇일까?

예상 1 : ...

< 함수 포인터 >

함수도 컴파일 -> 링크 과정에서 스택 메모리에 올라가게된다.

T (*p) (T val1 , T val2); // 라고 변수를 선언하고

Int Get_Max(int a, int b); // 라고 정의한 함수가있을때때

위와 같이 반환값과 매개변수가 정확히 일치할 때

함수 포인터로 사용할수있다.

P = Get_Max;로 초기화 // 함수명도 배열이름처럼 주소를 가지고 있다. 주소를 함수포인터가 가리킨다고 생각하자.

P(1,2); // 로 Get_Max(1,2)를 사용한 것처럼 사용이 가능

Why_함수명을 쓰면 되지 왜 굳이 저렇게 사용하지?

예를 들어서 템플릿 함수의 오버로딩으로 매개변수 타입에 따라 다른 함수를 작성하고 사용한다면 함수명을 재 정의하는 방식으로

직관성이 좋아질 수 있다. (직접 한번 구현해봐야... 명확히 안나옴

When_ sort(x,y,comp); 에서 comp 의 부분인데

Comp(a,b)//이런 형식이 아닌 함수 명을 쓰게 되어있다.

Sort()에서 //comp 는 함수의 주소를 받는 함수 포인터로 받는듯

Sort(T a,T b, Bool (*pt)(T,T)) //sort 원형 예상

단순히 직관성인가...

< Void 포인터 >

데이터 타입을 명시하지않은 포인터이다.

주의 : 간접 주소연산자를 사용할수없음

Why_ C++의 templete의 사용처럼 이식성이 좋음

Malloc은 이상하게 매번 알맞는 타입의 포인터를 반환해준다.

어떻게 이런 일이 가능한가?

How_ void* malloc(size_t size) ; // 동적 할당의 원형

바로 명시하지 않은 포인터를 반환하므로서

나중에 묵시적으로 캐스팅연산이 이루어지는 것이라 볼 수 있다.

C++로 넘어가게되면 위배되는 원칙인데.

데이터 타입을 엄격히 검사하겠다는 지침에 따라 New / Delete 로 바뀌게 된다.

< free >

Malloc 을 통해서 동적할당 받은 메모리는 명시적으로 해제해줘야한다. => 메모리 누수 발생위험

Free(arr); //

arr = NULL;

arr 의 메모리를 해제하지만 arr가 가리키는 주소는 여전하다.

=> arr = NULL; 명시도 반드시 해줘야함.

<& 참조자>

Why_ 포인터처럼 메모리에 접근할 수있으나 T const pt; 처럼 사용된다.(주소 값 변경 불가능)

How_ int & pt = a; // 선언과 동시에 초기화해야함

When_ void Swap(int & a, int& b){ … }

// &a,&b 주소연산자 용도로 쓰인것이 아닌 진짜 별명의 느낌으로

변수 명만 다를 뿐 실제론 Call By Reference 가 적용되어 실행 됨

<복사 생성자>

Why_ Car c1; Car C2(C1); 을하면 복사가 된다.

// C2 = C1; 으로 사용이 가능한데 어떻게 이루어지는 것 일까?

How_ 객체간 대입 연산이 이루어지게 하는 복사 생성자.

Car (const Car& obj)

: Speed(obj.Speed) , Limit(obj.Limit){};

직접 구현하지 않아도 컴파일러가 생성하는 디폴트 복사생성자 이다.

주의: 멤버변수에 동적 할당하는 변수가 있을 경우 얕은 복사가 진행된다.

Why_ 생성자 구문에서 동적 할당을하여 초기화를 진행할 텐데

대입연산을 할 경우 복사생성자가 호출되어 동적할당이 이루어지지않기 때문에 얕은 복사라고 하는 것이다.

복사 생성자에서도 동적할당을 하는 복사생성자를 오버 로딩하여 구현하면 깊은 복사라고 한다.

이 경우 직접 구현하여 동적 할당하는 변수는 동적할당하도록 해야함

<class 사용에서 동적할당을 하는 이유>

Why_

Car c1 ; 처럼 직접 사용할 수 있으나 이는 지역변수와 같으며

다른 함수에서 사용하는데 무리가있다.

필요할 때 사용하고 필요없을경우 메모리 할당을 해제해야 C언어의 강점이 드러난다.

How_ Car* C1 = new Car();// 생성 시

Delete C1; // 필요없을시

C1 = nullptr;

따라서 클래스를 동적할당하는 편이 메모리를 효율적으로 사용할 수 있다.

<임시 객체>

Why_

(s1 + s2)를 연산 후 임시로 담을 객체가 필요한데 임시로 사용하고 사용이 끝나면 메모리를 반환하게 된다.

String s1,s2; // s1 = "Hello" , s2 = "World " 로 가정

Const char* str = (s1 + s2).c_str();// 문자열로 캐스팅했을때.

Std:: cout << str ; // 컴파일 오류가 나게된다.

String + string 간의 Operator + 연산자 오버로딩으로 인해

String의 객체를 반환하는데 저장하질 않으면 반환된다.

내부적으로 Const char* str = (s1 + s2).c_str();

문장이 끝난 후 메모리 반환되기 때문에 의미 없는 주소를 가리키게된다.

<상속 관계에서 생성자와 소멸자>

Class Dog : class Animal

{

Class Dog()

: Animal() // 부모 생성자가 내부적으로 호출된다.

디폴트로 디폴트 생성자를 호출하기 때문에 매개변수가 필요한

생성자를 호출할 경우, 명시적으로 작성 해야함.

}

호출순서 : 부모생성자 -> 자식생성자

<상속 관계에서 소멸자>

부모 클래스에서는 자식클래스가 있는지 판명하기 어려운 구조라

만일 자식클래스를 업캐스팅 한 상황이라면 메모리 반환시에

자식클래스의 소멸자가 호출이 안될 경우가 존재한다.

따라서 소멸자 앞에 virtual ~Animal() // virtual 을 통해

자식 클래스가 존재함을 알리면 가상 테이블이 작성되므로

이를 통해 자식 클래스의 존재여부를 확인할수 있게된다.

< 업캐스팅 & 다운캐스팅 >

다형성을 위해 Animal arr[] = {Dog_class, Cat_class, ...};

등이 있다고 가정하자. 각 자식클래스는 업캐스팅을 하여 관리하는데

Arr[0].Sound() 를 호출 할경우 업캐스팅이 된 상황이기 때문에

Animal::Sound()가 호출이 된다.

이는 다형성의 의미를 잃어버려 상속의 의미가 사라진다.

함수 오버라이드를 이용하여 작성하면 Dog::Sound() , Cat::Sound()를 호출할수있다.

< Friend 메커니즘 >

상속관계에 있지않으나 외부 클래스에서 Private로 선언한 변수나 함수에 접근하고 싶은 경우가 존재한다.

그 경우를 위해 만들어진 것으로

Friend의 특징은 함수의 경우

함수원형은 클래스 내부에 선언되지만 본체는 클래스 외부에 작성된다.

```
Class Employee{
        Friend class Manager; // 프렌드 등록을 해두면면
        Private:   int price;   // Price 에 접근이 Manager 클래스에서 가능해진다.
}
```

< 연산자 오버로딩 >

Class 혹은 STL 간의 자료에 있어서 각 연산자들을 오버로딩하여 사용할수있다.

```
Class My_Vector { int x, int y;} // 각 0으로 초기화 하였다고 가정
Vector v1,v2;
Std::cin >> v1.x >> v1.y ; // 이렇게 입력을 받는 경우도
Std::Cin >> v1; 이런식으로  오버로딩을 통해 작성할수있다.
```

코드의 간결성을 늘릴수있고 직관적인 코딩이 가능하다.

<예외처리 >

프로그램을 작성하다 보면 에러가 발생하더라도 계속 실행이 되기를 원할 것이다.

If문을 이용해 모든 예외처리는 하기 쉽지 않고,

코드가 복잡해질 수 있다.

이를 위해 예외처리기를 개발하였고 이를 통해 쉬운 예외처리가 가능하다.

```
Try {
        if(~예외처리의 케이스 ) throw val;
        //문장1
}        //throw가 되지않으면 문장2는 실행되지 않는다.
  Catch(T val)
{//문장2}
```

예외처리를 사용하는 경우 함수원형을 선언할 때 throw 문구와 반환 타입을 적어주자.

```
// 예외처리로 무슨 데이터 타입의 throw를 날려주는지 표시
Int Get_max(int a, int b) throw (int); //무슨무슨경우 Throw
```

<static_Cast & dynamic_Cast> 이유

C++로 넘어오면서 데이터 타입에 엄격해지자는 목표가 있었는데 그 중 하나이다.

C언어의 문제점 예시 => int* ip; double* dp = (double*) ip;

//에러가 안났음.

Int와 double은 데이터표현이 다르기 때문에 값이 이상해짐

< static_Cast >

Int i_num = Static_cast<int>(d_num); 를 하면

데이터 타입을 검사하며 변환을 해준다.

<dynamic_Cast>

업 캐스팅 , 다운캐스팅처럼 객체를 대상으로 사용할 수도 있다.

문제점 : 객체로 넘어오면서 다운캐스팅 시에 문제가 있을 수 있는데.

예시 : Static_cast<Dog> (Cat);

// 다운캐스팅 변환에 실패하는 경우가 존재한다.

다운 캐스팅에대해 검사를 하여야하는데 Static같은 경우 검사를 하지않는다.

매번 안전한 객체가 들어가면 상관이 없겠지만 검사하는 데이터타입을 엄격히 검사하기 위해 등장했다.

Dynamic_cast는 다운캐스팅은 무조건 실패로 나오는데.

부모 객체에 virtual 가상함수가 있다면 가상테이블로 다운캐스팅이 가능한지 검사할 수 있다.

만일 변환에 실패한다면 아스키코드 '0' nullptr을

가리키기 때문에 변환실패도 검사할 수있다.

< 일반화 코딩의 핵심 템플릿>

제너릭 코딩은 코드의 중복을 막아줄수있고 코드의 크기를 줄이는데 도움이 많이된다.

만일 모든 데이터 타입에 대응해서 작성을 해야한다면 오버로딩을 통한 코드 중복이 많이 일어날것이다.

이를 없애기 위해 등장한 것이 템플릿 인데 템플릿으로 작성할 경우

해당 매개변수의 타입에 해당하는 템플릿 함수만 코드에 작성되기때문에

코드의 크기가 줄어드는 효과를 가져온다.

<템플릿 사용방법>

```
Template <typename T>
//컴파일러에게 먼저 템플릿 사용을 알린후
T Get_Max(T a ,T b){}
//   매개변수의 타입이 다른 경우
Template <typename T1,typename T2>
T Get_Max(T1 a, T2 b){}
// 매개변수로 배열이 전해졌을 경우
T Get_Max(int arr[], int arr2[]){};
// 매개변수를 명시하여 오버로딩 가능
```

<템플릿 클래스>

```
Template <typename T>
Class Box{
    Private:
        T data;
    Public :   Box();
};
```

< 템플릿의 Typedef 활용 >

Box<T>::Box(){...}// <T>를 사용하여야함.

typedef Box<int> ibox;//를 선언한후

Ibox box; 로 사용가능.

// 하지 않는다면

Box<int> box; 로 사용해야함

< STL > // 컨테이너의 종류

순차 컨테이너 : vector , list , deque

연관 컨테이너 : set , map , multiset &map 등

컨테이너 어뎁터 : Stack , queue , 우선순위 큐큐

컨테이너는 검증된 라이브러리이기때문에 사용하면 효율성을 높힐 수있다.

각 컨테이너 마다 장점과 단점이 명확하기 때문에 구분하여 사용하는 것이 중요하다

< vector >

vector는 배열의 메모리에 존재하는 연속성에서 오는 특징을위해 매우 중요하게 여긴 컨테이너이다.

Vector의 capacity를 초과할 때마다 메모리의 연속성을 유지하기위해 복사생성자를 호출하여 메모리의 탐색 및 새로운 할당이 실시된다.

=> 때문에 어느정도 데이터의 양이 예상이 된다면 미리 할당해주는 것이 좋다.

Why_ 중간에서의 데이터 삽입/삭제에 매우 약하다

메모리의 연속성을 유지해야하기 때문에 뒷 데이터들을 당겼다가

밀었다가 하는 작업이 일어나 느리게 작동하기 때문이다.

< list > singlelink list or Dlinklist

Vector와 다르게 메모리에 연속적으로 저장하지않고 각자 다음 pointer를 가지고있는

컨테이너이다.

논리적인 순차를 구현하여서 vector의 단점을 보완하였기

때문에 중간에서의 데이터 삽입/삭제가 빠르게 작동한다.

하지만 탐색에서 무조건 0 인덱스부터 탐색 해야하는 단점이 있다.

=> 배열과 호환이 안됨. [0] 접근이 불가능 하기때문.

이 단점은 크게 작용하는데.

이진 트리 탐색방법을 못하기 때문에 데이터가 크면

매우 느린 탐색속도가 일어나게된다.

< deque >

Vector 컨테이너와 매우 유사하지만 백터의 앞 인덱스로부터의

데이터 삽입.삭제가 가능한 컨테이너이다.

Deque<int> vec ;

Vec.push_front(int val); // vec.pop_front(); // 같이 사용한다.

<Map>

Key를 가지고 정렬을 한다. Key자체가 인덱스 의미를 갖기 때문에

추가는 오래 걸리지만 탐색이 매우 빠르다는 것이 특징이다.

<연관 컨테이너>

Set : Key가지고 정렬하는 케이스

MultiSet : 중복이 허용되는 set

Map : key - value를 가지고 정렬을하하는 형태

MultiSet: key가 중복될수있는 형태.

< Map >

Map<string,int> m;

Std::cin >> s ;

M[s]++; // 만약 s가 없었을경우우 자동으로 m에 요소가 추가되며

++로 인해 값이 추가된다.

For(Map<string,int>::iterator iter ; iter != m.end(); iter++)

// Iter->first || iter.second 로 key_value 접근이 가능하다

< 반복자 >

Iterator는 여러가지 컨테이너들을 다루기 쉽게 다음 요소로 향하는 pointer의 역할을 가지고 있다.

Iterator로 인해 다른 컨테이너에서도 일괄적으로 사용할 수 있는

일반화 프로그래밍을 구현하였고 내부적으로 template를 사용하여 구현한 것으로 보인다.

++연산자오버로딩을 통해 논리적 ,물리적으로 순차접근이 가능하다.

Iterator에서 컨테이너들이 가지는 객체의 end()는 실제로 데이터가 들어있지 않으므로 null을 갖기에 사용에 유의 하여야한다.

Iter 끼리의 대소구분과 ++ 증감 연산자는 사용이 가능하다.

<컨테이너 어뎁터 > // stack & queue

이미 존재하는 컨테이너로 새로운 기능을 제공하기위해 만들어짐

Stack 은 회문을 검사하거나 자료형을 입력과 출력을 거꾸로 하고싶을때 사용함.

Queue는 BFS에서 사용

우선순위 큐는 탐색할 때 최소힙 이나 최대힙힙을 이용해 이분탐색을 한다.

=> 때문에 데이터의 량의 탐색에 쓰인다.

< Algorithm 헤더파일 >

코딩테스트에서 매우 효과적으로 사용될 헤더파일이다.

컨테이너의 어댑터는 입출력에 관련이 있었다고 한다면

컨테이너의 알고리즘을 담당하는 헤더파일이다.

변경 알고리즘과 비변경 알고리즘이 존재한다.

변경 알고리즘 : 컨테이너의 요소를 바꾸는 알고리즘

비변경 알고리즘 : 컨테이너의 요소는 변경하지 않는 알고리즘

< 비변경 알고리즘 >

계수 알고리즘

Count(v.begin(), v.end(), find_val);

// v컨테이너의 범위 에서 val값을 갖는 요소 개수를 반환해준다.

Count_if(v.begin(), v.end(), 함수명(함수포인터));

// 함수의 조건에서 bool을 반환하는 함수를 작성하여 함수명을 적어준다.

위 함수에서 반환하는 bool 값에 참이 되는 값을 count해준다.

탐색 알고리즘

Search() // 대상 범위 내에서 지정된 시퀀스의 요소와 동일한 첫 번째 시퀀스를 검색합니다.

Search_n() // 범위에서 특정 값의 요소가 지정된 수만큼 있거나 이진 조건자가 지정한 해당 값과 관련이 있는 첫 번째 하위 시퀀스를 검색합니다.