



C++ - 모듈 08

템플릿화된 컨테이너, 이터레이터, 알고리즘

요약:

이 문서에는 C++ 모듈 중 모듈 08의 연습 문제가 포함되어 있습니다.

버전: 7

콘텐츠

I	소개	2
II	일반 규칙	3
III	모듈별 규칙	5
IV	연습 00: 쉬운 찾기	6
V	연습 01: 스펀	7
VI	연습 02: 돌연변이 홍물	9

1장 소개

C++는 비야른 스트로스트룹이 C 프로그래밍 언어 또는 '클래스가 있는 C'의 변형으로 만든 범용 프로그래밍 언어입니다(출처: [위키백과](#)).

이 모듈의 목표는 **객체 지향 프로그래밍**을 소개하는 것입니다. 이것이 C++ 여정의 시작점이 될 것입니다. OOP를 배우기 위해 많은 언어가 권장됩니다. C++는 여러분의 오랜 친구인 C에서 파생된 언어이기 때문에 C++를 선택하기로 결정했습니다. C++는 복잡한 언어이며, 코드를 단순하게 유지하기 위해 C++98 표준을 준수합니다.

최신 C++는 여러 측면에서 많이 다르다는 것을 알고 있습니다. 따라서 능숙한 C++ 개발자가 되고 싶다면 42개의 공통 코어를 넘어서는 것은 여러분에게 달려 있습니다!

2장 일반 규칙

컴파일

- c++와 -Wall -Wextra -Werror 플래그를 사용하여 코드를 컴파일합니다.
- 플래그 -std=c++98을 추가하면 코드가 계속 컴파일됩니다.

서식 및 이름 지정 규칙

- 연습 디렉터리의 이름은 ex00, ex01, ..., exn
- 가이드라인에서 요구하는 대로 파일, 클래스, 함수, 멤버 함수 및 속성의 이름을 지정하세요.
- 클래스 이름을 **대문자 대소문자** 형식으로 작성합니다. 클래스 코드가 포함된 파일은 항상 클래스 이름에 따라 이름이 지정됩니다. 예를 들어 ClassName.hpp/ClassName.h, ClassName.cpp 또는 ClassName.hpp. 예를 들어, 벽돌 벽을 의미하는 "BrickWall" 클래스의 정의가 포함된 헤더 파일이 있다면 그 이름은 BrickWall.hpp가 됩니다.
- 달리 지정하지 않는 한, 모든 출력 메시지는 새 줄 문자로 끝나야 하며 표준 출력에 표시되어야 합니다.
- *안녕, 노미네트!* C++ 모듈에는 코딩 스타일이 강제되지 않습니다. 좋아하는 스타일을 따를 수 있습니다. 하지만 동료 평가자가 이해할 수 없는 코드는 채점할 수 없는 코드라는 점을 명심하세요. 깔끔하고 읽기 쉬운 코드를 작성하기 위해 최선을 다하세요.

허용/금지

더 이상 C로 코딩하지 마세요. 이제 C++로 전환하세요! 그러므로:

- 표준 라이브러리의 거의 모든 것을 사용할 수 있습니다. 따라서 이미 알고 있는 것을 고수하는 대신 익숙한 C 함수의 C++ 버전을 최대한 많이 사용하는 것이 현명할 것입니다.
- 하지만 다른 외부 라이브러리는 사용할 수 없습니다. 즉, C++11(및 파생 형식) 및 Boost 라이브러리는 금지됩니다. 다음 함수도 금지됩니다: `*printf()`, `*alloc()` 및 `free()`. 이 함수를 사용하면 성적은 0점이 됩니다.

- 명시적으로 달리 명시되지 않는 한 네임스페이스 `<ns_name>` 및 친구 키워드는 금지되어 있습니다. 그렇지 않으면 성적은 -42점이 됩니다.
- **모듈 08과 09에서만 STL을 사용할 수 있습니다.** 즉, 그때까지는 컨테이너(벡터/리스트/맵 등)와 알고리즘(<알고리즘> 헤더를 포함해야 하는 모든 것)을 사용할 수 없습니다. 그렇지 않으면 성적이 -42점이 됩니다.

몇 가지 설계 요구 사항

- 메모리 누수는 C++에서도 발생합니다. 메모리를 할당할 때(새로운 키워드)의 경우 **메모리 누수**를 방지해야 합니다.
- 모듈 02부터 모듈 09까지, **명시적으로 달리 명시된 경우를 제외하고** 클래스는 **정통 정석 양식**으로 디자인해야 합니다.
- 헤더 파일에 있는 모든 함수 구현(함수 템플릿 제외)은 연습에 0을 의미합니다.
- 각 헤더를 다른 헤더와 독립적으로 사용할 수 있어야 합니다. 따라서 필요한 모든 종속성을 포함해야 합니다. 그러나 **include 가드**를 추가하여 이중 포함 문제를 피해야 합니다. 그렇지 않으면 성적이 0점이 됩니다.

읽기

- 필요한 경우 추가 파일을 추가할 수 있습니다(예: 코드를 분할하는 경우). 이러한 과정은 프로그램에서 확인하지 않으므로 필수 파일을 제출하는 한 자유롭게 추가할 수 있습니다.
- 때때로 연습 지침이 짧아 보이지만 지침에 명시되지 않은 요구 사항이 예제에 나와 있는 경우가 있습니다.
- 시작하기 전에 각 모듈을 완전히 읽으세요! 정말 그렇게 하세요.
- 오딘의, 토르의! 머리를 써라!!!



많은 클래스를 구현해야 합니다. 자주 사용하는 텍스트 편집기를 스크립팅할 수 없다면 이 작업이 지루해 보일 수 있습니다.



운동을 완료하는 데는 어느 정도의 자유가 주어집니다. 그러나 필수 규칙을 따르고 게으르지 마세요. 유용한 정보를 많이 놓칠 수 있습니다! 이론적 개념에 대해 주저하지 말고 읽어보세요.

제3장

모듈별 규칙

이 모듈에서는 표준 컨테이너와 표준 알고리즘 없이도 연습 문제를 풀 수 있음을 알 수 있습니다.


그러나 **이를 사용하는 것이 바로 이 모듈의 목표**입니다. STL을 사용할 수 있습니다. 예, **컨테이너**(벡터/리스트/맵 등)와 **알고리즘**(헤더 <알고리즘>에 정의됨)을 사용할 수 있습니다. 또한 가능한 한 많이 사용해야 합니다. 따라서 적절한 곳에 적용하기 위해 최선을 다하세요.

코드가 예상대로 작동하더라도 그렇지 않으면 매우 나쁜 성적을 받게 됩니다. 게으르지 마세요.

평소처럼 헤더 파일에서 템플릿을 정의할 수 있습니다. 또는 원하는 경우 헤더 파일에 템플릿 선언을 작성하고 그 구현을 .tpp 파일에 작성할 수 있습니다. 어떤 경우든 헤더 파일은 필수이며 .tpp 파일은 선택 사항입니다.

제4장

연습 00: 쉬운 찾기

	운동 : 00
	쉬운 찾기
제출 디렉토리 : <i>ex00/</i>	
제출할 파일 : 메이크파일, <i>main.cpp</i> , <i>easyfind.{h, hpp}</i> 및 선택 파일: <i>easyfind.tpp</i>	
금지된 기능 : 없음	

첫 번째 쉬운 운동은 올바른 출발을 위한 방법입니다.

유형 `T`를 허용하는 함수 템플릿 `easyfind`를 작성합니다. 두 개의 매개변수가 필요합니다.

첫 번째는 유형이 `T`이고 두 번째는 정수입니다.

T가 정수의 컨테이너라고 가정하면, 이 함수는 첫 번째 매개변수에서 두 번째 매개변수의 첫 번째 발생을 찾아야 합니다.


발견되지 않으면 예외를 발생시키거나 원하는 오류 값을 반환할 수 있습니다. 영감이 필요하다면 표준 컨테이너가 어떻게 동작하는지 분석해 보세요.

물론 모든 것이 예상대로 작동하는지 확인하기 위해 자체 테스트를 구현하고 제출하세요.



연관 컨테이너를 처리할 필요가 없습니다.

5장 연습 01: 스펠

	운동 : 01
Span	
체크인 디렉토리 : <i>ex01/</i>	
제출할 파일 : <i>Makefile, main.cpp, Span.{h, hpp}, Span.cpp</i>	
금지된 기능 : 없음	

최대 N개의 정수를 저장할 수 있는 **Span** 클래스를 개발합니다. N은 부호 없는 int 변수이며 생성자에 전달되는 유일한 매개변수입니다.

이 클래스에는 스펠에 단일 숫자를 추가하는 `addNumber()`라는 멤버 함수가 있습니다. 이 함수는 스펠을 채우는 데 사용됩니다. 이미 N개의 요소가 저장되어 있는 상태에서 새 요소를 추가하려고 하면 예외가 발생합니다.

다음으로, 두 개의 멤버 함수, 즉 `shortestSpan()` 및 `longestSpan()`을 구현합니다.

이 함수는 각각 저장된 모든 숫자 사이의 최단 범위 또는 최장 범위(또는 원하는 경우 거리)를 찾아서 반환합니다. 저장된 숫자가 없거나 하나만 있으면 스펠을 찾을 수 없습니다. 따라서 예외를 던집니다.

물론 직접 테스트를 작성하면 아래 테스트보다 훨씬 더 철저할 것입니다. 최소 10,000개의 숫자로 스펠을 테스트하세요. 더 많으면 더 좋습니다.

이 코드를 실행합니다:

```
int main()
{
    스패 sp = 스패(5);

    sp.addNumber(6);
    sp.addNumber(3);
    sp.addNumber(17);
    sp.addNumber(9);
    sp.addNumber(11);

    std::cout << sp.shortestSpan() << std::endl;
    std::cout << sp.longestSpan() << std::endl;

    반환값은 0입니다;
}
```

출력해야 합니다:

```
$> ./ex01
2
14
$>
```


마지막으로, **다양한 이터레이터를** 사용해 스패를 채우면 좋을 것입니다. addNumber()를 수천 번 호출하는 것은 매우 성가신 일입니다. 멤버 함수를 구현 하여 한 번의 호출로 스패에 많은 숫자를 추가할 수 있습니다.



잘 모르겠다면 컨테이너를 공부하세요. 일부 멤버 함수는 컨테이너에 요소 시퀀스를 추가하기 위해 다양한 반복자를 사용합니다.

제6장

연습 02: 돌연변이 혐오물

	운동 : 02
돌연변이 혐오물	
제출 디렉토리 : <i>ex02/</i>	
제출할 파일 : 메이크파일, <code>main.cpp</code> , <code>MutantStack.{h, hpp}</code> 및 옵션 파일: <code>MutantStack.tpp</code>	
금지된 기능 : 없음	

이제 더 심각한 문제로 넘어갈 차례입니다. 이상한 것을 개발해 봅시다.

`std::stack` 컨테이너는 매우 훌륭합니다. 안타깝게도 이 컨테이너는 이터러블하지 않은 유일한 STL 컨테이너 중 하나입니다. 안타깝네요.

하지만 왜 우리가 이것을 받아들여야 할까요? 특히 원래 스택을 자유롭게 도살하여 누락된 기능을 만들 수 있다면 더욱 그렇습니다.

이 불공평함을 바로잡으려면 `std::stack` 컨테이너를 이터러블로 만들어야 합니다.

MutantStack 클래스를 작성합니다. 이 클래스는 `std::stack`으로 구현됩니다. 모든 멤버 함수와 추가 기능인 **이터레이터**를 제공합니다.

물론 모든 것이 예상대로 작동하는지 확인하기 위해 직접 테스트를 작성하고 제출해야 합니다.

아래에서 테스트 예시를 찾아보세요.

```
int main()
{
    돌연변이 스택<int>    mstack;

    mstack.push(5);
    mstack.push(17);

    std::cout << mstack.top() << std::endl;

    mstack.pop();

    std::cout << mstack.size() << std::endl;

    mstack.push(3);
    mstack.push(5);
    mstack.push(737);
    //[...]
    mstack.push(0);

    MutantStack<int>::이터레이터 it = mstack.start();
    MutantStack<int>::이터레이터 ite = mstack.end();

    ++it;
    --그것;
    동안 (it != ite)
    {
        std::cout << *it << std::endl;
        ++it;
    }
    std::stack<int> s(mstack);
    반환값은 0입니다;
}
```

한 번은 돌연변이 스택으로 실행하고, 두 번째는 돌연변이 스택을 예를 들어 `std::list`로 대체하여 실행하면 두 출력은 동일해야 합니다. 물론 다른 컨테이너를 테스트할 때는 해당 멤버 함수를 사용하여 아래 코드를 업데이트하세요(`push()`는 `push_back()`이 될 수 있습니다).

