

STL - 2주 2일

Wulong

실행 시간 측정

프로그램의 실행 시간을 측정하는 방법을 배워봅시다. 어떤 알고리즘이 다른 알고리즘보다 더 좋다고 말할 때의 비교기준은 원하는 결과를 얻는데 걸리는 시간과 사용된 자원(메모리)입니다. 시간과 자원은 서로 반비례합니다. 시간이 더 적게 걸리면서 자원도 덜 사용할 수 있다면 정말 좋겠지만 그렇지 않습니다. 한쪽이 좋으면 다른 쪽은 나쁩니다.

다음은 3초간 시간을 쓰는 코드입니다.

```
#include <iostream>
#include <thread>
using namespace std;

int main()
{
    cout << "시작" << endl;

    this_thread::sleep_for( 3s );           // 3초를 쉬는 코드

    cout << "3초가 지남" << endl;
}
```

실제로 3초가 걸리는지 알고 싶습니다. 3초가 걸리는 코드 시작 전에 스톱워치의 시작 버튼을 누르고 코드가 끝난 후 다시 버튼을 누르면 3초를 잴 수 있을 겁니다. C++에서는 다음과 같이 코딩합니다.

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;

int main()
{
    cout << "시작" << endl;

    chrono::steady_clock::time_point start = chrono::steady_clock::now();
                                                // 시간값을 얻는다

    this_thread::sleep_for( 3s );           // 3초를 쉬는 코드
```

```

    chrono::steady_clock::time_point end = chrono::steady_clock::now();
                                   // 시간값을 다시 얻는다

    chrono::duration duration = end - start;    // 시간 간격을 잰다

    chrono::duration d = chrono::duration_cast<chrono::seconds>(duration);
                                   // 표현하고 싶은 단위로 간격을 변환한다

    cout << d.count() << "초가 걸림" << endl;

    cout << "3초가 지남" << endl;
}

```

시간 관련 함수는 <chrono>에 있습니다. 프로그램에서 시간을 이용할 일은 두 가지가 있습니다.

- 현재 시간값을 알아낸다. (예: 2020년 3월 23일 월요일 10시 10분)
- 경과한 시간을 잰다.

시간값을 알아내는 것은 나중에 save 파일 작성에서 설명하겠습니다.

위에 적은 코드는 이해를 돕기 위해 최대한 상세하게 늘어 쓴 것입니다. 어디에서도 이와 같은 내용을 찾아보기 어려울 겁니다. 나도 찾고 이해해서 이런 코드를 적기까지 고생했던 기억이 있습니다. 실제로는 이렇게 장황하게 쓸 필요도 이유도 없습니다. C++ 언어의 기능을 최대한 이용해서 간단하게 고쳐보겠습니다.

```

#include <iostream>
#include <thread>
#include <chrono>
using namespace std;

int main()
{
    cout << "시작" << endl;

    using namespace std::chrono;

    auto start = steady_clock::now();    // 시간값을 얻는다

    this_thread::sleep_for( 3000ms );    // 3초를 쉬는 코드

    auto d = duration_cast<milliseconds>(steady_clock::now() - start);

    cout << d.count() << "밀리초가 걸림" << endl;
}

```

어떻습니까? 마음에 드나요? 시간을 재고 싶은 코드 앞뒤로 빨간색 코드를 붙이기만 하면 간단하게 시간을 측정할 수 있습니다. 언제든지 필요할 때 붙여 넣어 시간을 재 주세요. 몇 가지 정리하겠습니다.

- auto는 이럴 때 사용합니다.
- 3초를 표현할 때 3s라고 했는데요. 이때 s를 사용자 정의 리터럴이라고 합니다. 순전히 프로그래머의 편의를 위한 것입니다. 시간을 잴 때 1000분의 1초인 ms를 주로 사용합니다.
- 시간을 나노초까지 잴 수 있습니다만 의미가 없습니다. ms는 알고리즘 시간을 측정해서 비교할 수 있는 의미있는 정확도가 있습니다.
- using namespace std::chrono;와 같이 줄여 쓰지는 않습니다.

시간을 잴 수 있게 되었으니 지난 시간 공부한 sort 함수에 얼마나 많은 시간이 걸리는 지 재 보겠습니다.

```
#include <iostream>
#include <chrono>
#include <algorithm>
#include <random>
using namespace std;

int main()
{
    default_random_engine dre;
    uniform_int_distribution<> uid;

    int data[10'0000];

    for (int& d : data)
        d = uid( dre );

    auto start = chrono::steady_clock::now();

    sort( begin(data), end(data) );

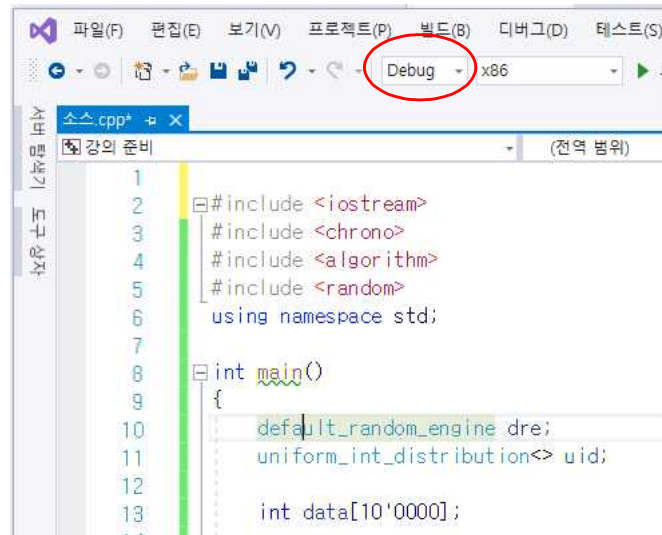
    auto d = chrono::duration_cast<chrono::milliseconds>(chrono::steady_clock::now() -
start);

    cout << d.count() << "밀리초가 걸림" << endl;

    // 정렬되었나 확인
    cout << data[0] << " --- " << data[10'0000 - 1] << endl;
}
```

10만개의 정수를 오름차순으로 정렬하는데 내 컴퓨터는 11ms가 걸렸습니다. 내 컴퓨터는 낫습니다. 여러분 것은 훨씬 더 빠른 속도를 보일 것입니다. 혹시 100ms 정도로 10배 이상

느리게 나온 학생은 비주얼 스튜디오 화면을 확인해 보세요.



디버그모드를 사용하면 프로그램이 형편없이 느리게 실행된다는 것을 잊지 마세요. 동일한 자료를 qsort로 정렬하면 시간은 얼마나 소요될까요? 더 빠를까요 아니면 느릴까요? 궁금하면 바로 실행해서 비교해 볼 수 있습니다. 복습할 겸 얼른 해 보세요. 5분이면 비교해 볼 수 있을 겁니다.

템플릿과 제네릭 프로그래밍

제네릭 프로그래밍이란 자료형과 무관한 프로그래밍을 말하는 것입니다. 코드로 설명하는 것이 이해하기 쉽겠습니다. 1주 1일 강의의 change 함수입니다. 이 함수는 두 int의 값을 서로 바꿉니다.

```
#include <iostream>
using namespace std;
```

```
void change( int&, int& );
```

// 함수의 선언

```
int main()
{
    int a { 1 };
    int b { 2 };
```

```

    change( a, b );

    cout << a << ", " << b << endl;
}

void change( int& a, int& b )           // 함수의 정의
{
    int t { a };

    a = b;

    b = t;
}

```

C++ 언어에는 몇 종류의 자료형이 있습니까? char, int, double, short*, long, long long, long double. 세어보니 십 여 가지 되려나요? 정말 그렇습니까? 자료형의 기본 목적은 무엇입니까? 컴퓨터에게 필요한 만큼 메모리를 달라는 것이죠.

위에 나열한 1, 2, 4, 8 바이트만 사용해서 프로그래밍할 수 있다면 얼마나 좋겠습니까? 우리는 새 자료형을 마음대로 만들어 얼마든지 더 많은 메모리를 요구할 수 있습니다. 키워드 class 를 사용해서 이렇게 할 수 있습니다. 다시 물어보겠습니다,

C++ 언어에는 몇 종류의 자료형이 있습니까?

C++의 자료형 갯수는 무한합니다. 사용자가 마음대로 자료형을 만들어 낼 수 있습니다. 다시 change 함수로 돌아가 보겠습니다. string 두 개의 값을 교환하고 싶습니다. change 함수를 다시 선언하고 정의하면 그렇게 할 수 있습니다.

```

// 생략

void change( int&, int& );           // 함수의 선언
void change( string&, string& );    // 함수의 선언

int main()
{
}

void change( string& a, string& b )  // 함수의 정의
{
    string t{ a };
    a = b;
    b = t;
}

```

함수의 본문이 크게 변한 것은 없어 보입니다. 바뀌어야 할 자료형만 달라졌습니다. 자료형의 갯수는 무한합니다. 어떤 것을 바꾸게 될지는 모릅니다. 필요할 때 마다 change 함수를 새로 만들 수는 없는 노릇입니다. 사람이 할 짓이 아닙니다. 컴퓨터 시켜야 하겠습니다.

필요한 자료형을 알려주면 컴퓨터가 대신 소스코드를 만들게 할 수 있습니다.

키워드 **template**이 이런 일을 합니다. 아래 프로그램을 타이핑해서 실행시켜 보세요.

```
#include <iostream>
#include <string>
using namespace std;

template <typename T>
void change( T&, T& );           // 함수 템플릿 선언

int main()
{
    string a { "19"s };
    string b { "corona" };

    change( a, b );
    // change<string>( a, b );    // 정식 사용법

    cout << a << b << " 바이러스" << endl;
}

template <typename T>
void change( T& a, T& b )       // 함수 템플릿 정의
{
    T t{ a };
    a = b;
    b = t;
}
```

먼 길을 돌아 왔습니다. 템플릿은 C++ 언어 강의에서 학기가 끝날 때 소개하고 지나가는 내용이라 잘 생각나지 않을 수 있습니다. 그렇지만 이번 강의로 이해하는 데 무리가 없을 것입니다. 필요성을 충분히 설명하려고 노력했습니다. 더 상세한 내용을 공부해 나가는 데 큰 문제는 없을 것이라고 안심시키고 싶지만 그럴 수는 없습니다. 템플릿은 만만한 내용이 절대 아닙니다. C++ 언어의 최신 표준에서 제일 변화가 많은 부분이 바로 템플릿으로 이루어진 STL입니다. 조금이라도 공부를 게을리 하면 외계어 같은 모습에 좌절하게 됩니다. 다행히도 템플릿으로 코딩해야 하는 소수의 전문가가 아니라면 진짜 어려운 부분을 모르더라도 효율적인 C++ 프로그램을 작성하는데는 큰 문제가 없습니다. 정말 다행입니다.

C++의 템플릿을 따라올 수 있는 언어는 없습니다. 몇몇 언어가 흉내만 낼 뿐입니다. 고맙게도 C++ 언어를 잘 정리해서 설명한 자료가 있어 소개해 봅니다.

<https://namu.wiki/w/C%2B%2B>. 꼭 읽어 보시기 바랍니다. 10분 내에 못 읽겠지만 10분 쉬겠습니다.

위의 함수 템플릿 `change`는 어떤 자료형이더라도 함수에 전달되는 두 개의 인자가 같은 자료형이기만 하면 두 자료의 내용을 바꿀 수 있는 `change` 함수를 만들어 냅니다. 이렇게 자료형에 무관한 함수를 제네릭 함수라고 하는 것입니다. 정말 중요하기 때문에 계속 반복하고 있습니다.

템플릿은 두 가지가 있습니다.

- 함수를 만들어 내는 함수 템플릿
- 자료형을 만들어 내는 클래스 템플릿

STL은 자료구조와 알고리즘을 제공하는 C++ 언어의 라이브러리입니다.

STL에서 제공하는 자료구조는 모두 클래스 템플릿으로 작성되어 있습니다.

STL에서 제공하는 알고리즘은 모두 함수 템플릿으로 작성되어 있습니다.

STL의 T가 바로 Template입니다. 모두 템플릿으로 이루어진 표준 라이브러리(Standard Library)입니다. 온통 템플릿 세상입니다. 템플릿을 잘 알수록 깊고 자세하게 이해할 수 있습니다. 클래스 템플릿을 설명하고 과제를 내며 이번 강의를 마치겠습니다.

Array 템플릿

배열은 단순한 자료구조입니다. 동일한 자료형을 물리적으로 연속된 공간에 저장합니다. `int`를 예로 들어 `int`형 10개를 배열에 저장해 보겠습니다.

```
int a[ 10 ] { 1, 2, 3, 4, 5 };
```

배열 이름 `a`는 40 바이트(`sizeof(int) * 10`) 메모리의 시작 번지를 가리키는 포인터입니다. 메모리가 실제로 붙어있으므로 `a`만 기억하고 있으면 배열의 다른 요소로 이동하는 것은 $O(1)$ 의 시간복잡도를 갖습니다. `a[0]`에 액세스하나 `a[9]`에 액세스하나 걸리는 시간은 같다는 것입니다. 자료에 접근하는 속도면에서 이 보다 빠른 자료구조는 없습니다. 그런데 언어가 기본으로 제공하는 배열은 시작번지만 기억하기 때문에 정해진 메모리의 경계를 벗어난 읽고 쓰기를 막을 방법이 없습니다.

```
int a[ 10 ] { 1, 2, 3, 4, 5 };
```

```
cout << a[-1] << a[10] << a[100] << endl; // 이 코드도 문제 없다.
```

필요하다면 언제든지 경계를 벗어난 접근을 막기 위해 배열과 동일한 인터페이스를 제공하면서 경계를 검사하는 새로운 자료구조를 만들 수 있습니다. 새로운 클래스를 만들면 됩니다. 우리도 만들어 보겠습니다.

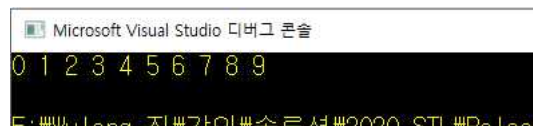
```
#include <iostream>
using namespace std;

int main()
{
    IntArray a( 10 ); // 정수 10개를 담을 메모리 확보

    for (int i = 0; i < 10; ++i)
        a[i] = i; // a에 쓰기

    for ( int i = 0; i < 10; ++i )
        cout << a[i] << ' '; // a에서 읽기
    cout << endl;
}
```

[실습] main()이 문제없이 실행되도록 class를 정의하라. 실행하면 다음과 같이 화면 출력되어야 한다.



실행에 문제가 없도록 하라고 했으니 되도록 간단하게 만들면 되겠습니다. 객체 a를 생성할 때 배열 기호 []를 사용할 수는 없습니다. 이것만 빼면 나머지는 배열과 같습니다. 나는 여러분이 복습 할 수 있도록 조금 더 붙인 코드를 만들어 봤습니다.

```
class IntArray {
    size_t num { 0 };
    int* data{ nullptr };

public:
    explicit IntArray( size_t n ) : num { n }, data { new int[ num ] } { }
```



```

~IntArray() {
    delete[] data;
}

IntArray( const IntArray& ) = delete;
IntArray& operator=( const IntArray& ) = delete;

int operator[]( int idx ) const {
    return data[idx];
}

int& operator[]( int idx ) {
    return data[idx];
}
};

```

원하는 기능을 구현했습니다. 필요성 설명은 경계를 검사하는 배열에서 시작했지만 이 클래스는 그렇게 하지는 않았습니다. 지금은 자료형에 집중하는 것이 중요하기 때문입니다. STL이 경계검사를 어떤 철학으로 하는 지는 나중에 설명하게 될 것입니다.

앞에서 예로 들었던 함수 템플릿 change의 설명과 마찬가지로 int만을 배열로 사용하지는 않습니다. 어떠한 자료형이더라도 배열로 만들 수 있습니다. 같은 이유로 class IntArray도 템플릿으로 만들 수 있습니다.

[2주 2일 - 과제]

```

#include <iostream>
using namespace std;

int main()
{
    Array<int> a( 10 );           // 정수 10개를 담을 메모리 확보

    for (int i = 0; i < 10; ++i)
        a[i] = i;               // a에 쓰기

    for ( int i = 0; i < 10; ++i )
        cout << a[i] << ' ';   // a에서 읽기
    cout << endl;
}

```

[문제 1] main()이 문제없이 실행되도록 클래스 템플릿을 정의하라.

위 문제를 해결하는데 큰 어려움은 없을 것입니다. 클래스 템플릿으로 객체를 만들 때는 함수 템플릿과는 다르게 <int>를 생략할 수 없습니다. 함수는 인자로부터 자료형을 유추할 수 있지만 클래스는 그럴 수 없기 때문입니다.

STL에는 배열을 확장한 array라는 클래스 템플릿으로 작성된 자료구조가 있습니다. 그런데 [문제 1]과 같이 int 배열을 만들려면 다음과 같은 문법으로 객체를 만들어야 합니다.

```
#include <array>

array<int, 10> a;           // int 10개 배열 a
```

배열의 원소 갯수를 의미하는 숫자 10을 템플릿의 인자로 전달하였습니다. 이렇게 자료형이 아닌 숫자 10을 non-type template parameter라고 부릅니다.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    Array<string, 10> a;           // string 10개를 담을 수 있는 자료구조 a

    for (int i = 0; i < 10; ++i)
        a[i] = "string"s + to_string( i );    // a에 쓰기

    for ( int i = 0; i < 10; ++i )
        cout << a[i] << endl;                // a에서 읽기
}
```

[문제 2] main()이 문제없이 실행되도록 클래스 템플릿을 정의하라.

문제를 해결하지 못한 학생은 무엇을 모르는지 자세하게 적어 제출하면 되겠습니다.

2주 1일과 2주 2일 과제를

문서 파일 하나로 만들어 (형식은 윈도우 텍스트 문서 형식으로)

3주 1일 강의 시작 전까지 e-class로 제출해 주세요.

과제는 친구들과 의논하여 해결하는 것을 권장합니다.

과제를 평가하여 앞으로 얼마나 공부해야 할 것인지 알려주겠습니다.