STL - 7주 2일

Wulong

제네릭 함수

STL 알고리즘 함수는 반복자를 인자로 받습니다. 코딩하면서 설명하는 것이 좋겠습니다. 컨테이너의 원소를 화면에 출력하는 함수 print를 만들어 보겠습니다.

```
#include <iostream>
#include "String.h"
using namespace std;

int main( )
{
    String str { "The quick brown fox jumps over the lazy dog" };

    print( str.begin( ), str.end( ) );
}
```

이 프로그램이 문제없이 실행되도록 print를 선언하고 정의하는 것이 문제입니다. 문제를 해결하는 것은 간결할수록 좋습니다. 나는 print를 이렇게 선언하고 정의하겠습니다.

```
// declaration
void print( String::iterator b, String::iterator e );

// definition
void print( String::iterator b, String::iterator e )
{
    while ( b != e ) {
            cout << *b << ' ';
            ++b;
    }
    cout << endl;
}
```

이것이 가장 간결한 답입니다. 함수의 인자형식이 정확하게 맞습니다. print 함수에 전달된 인자는 반복자입니다. String의 반복자는 !=, *, ++ 연산을 지원하므로 문제없이 컴파일되고 실행됩니다. 원소와 원소 사이 출력에는 빈칸을 삽입해 봤습니다.

한 번 물어 보겠습니다. 이 print 함수는 제네릭 함수입니까?

아닙니다. 이 print 함수는 String::iterator가 전달될 때만 동작하기 때문에 제네릭 함수가

아닙니다. 그런데 이 함수의 동작을 살펴보니 forward 반복자가 전달되기만 한다면 아무런 문제없이 출력기능을 할 수 있을 것으로 보입니다.

문제를 조금 바꿔 보겠습니다.

[문제] print 함수에 전달되는 인자의 자료형이 무엇인지 모르겠지만 전달된 인자가 !=, *, ++ 연산을 지원하기만 한다면 그 값을 화면에 출력하도록 프로그램하라.

이렇게 문제를 바꾸면 print 함수는 제네릭 함수가 되어야 합니다. C++ 언어에서 제네릭을 구현하는 키워드는 **template**입니다. 다음과 같이 함수 template을 작성하면 됩니다.

```
// declaration
template <class T>
void print( T b, T e );

// definition
template <class T>
void print( T b, T e )
{
    while ( b != e ) {
            cout << *b << ' ';
            ++b;
    }
    cout << endl;
}
```

컴파일러는 print 함수 템플릿으로 실제 함수를 만들어 오류가 없는지 검사하고 실행되는 코드를 생성합니다. print는 forward 반복자의 기능만 만족한다면 무엇이 전달되어도 상관없습니다. print는 자료가 어디에 저장되어 있는지 어떤 컨테이너의 원소를 출력해야 하는 것인지 알 수 없으며 관심도 없습니다.

제네릭 함수가 있다면 다음과 같은 프로그램을 자유롭게 쓸 수 있습니다.

```
int main( )
{
    String str { "The quick brown fox jumps over the lazy dog" };
    vector<char> v { str.begin(), str.end() };
    list<char> l { v.begin( ), v.end( ) };

    print( str.begin( ), str.end( ) );
    print( v.begin( ), v.end( ) );
    print( l.begin( ), l.end( ) );
}
```

print에 전달되는 인자의 형식이 모두 다르다는 것을 살펴보세요.

모든 STL 알고리즘 함수가 이렇게 template으로 작성되어 있습니다. 몇 몇 함수를 빼고는 구현도 크게 어렵지 않습니다. 간단한 함수를 살펴보면서 제네릭 함수가 어떻게 코딩되어 있는지 알아보겠습니다.

다음은 반복자로 주어진 [first, last)구간에서 가장 작은 값과 가장 큰 값을 찾아주는 STL 알고리즘 함수인 **minmax_element**입니다. 아래에 보인 것은 minmax_element의 4가지 오버로딩 버전 중에서 가장 간단한 함수의 선언문입니다. 함수의 이름에서 기능을 알 수 있도록 작명하였음을 알 수 있습니다. 표준함수는 모두 소문자로 시작합니다. 두 개의 값을 return 해야 하기 때문에 **pair**로 return 합니다. 함수의 인자는 forward 반복자의 요구사항을 준수하면 됩니다.

```
template<class ForwardIterator>
constexpr pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last);
```

다음과 같이 사용합니다.

```
int main( )
{
    vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    pair<vector<int>::iterator, vector<int>::iterator> p
            = minmax_element( v.begin( ), v.end( ) );

    cout << "최소: " << *p.first << ", 최대: " << *p.second << endl;
}
```

실행 결과입니다.

리턴 값 p는 first와 second라는 이름의 멤버 변수를 갖는 구조체 **pair**입니다. 진짜 프로그램에서 이렇게 쓰는 사람 아무도 없습니다.

```
    auto p = minmax_element( v.begin( ), v.end( ) );
```

이럴 때 auto를 사용한다니까요. 훨씬 간결합니다. 의미도 명확합니다. STL을 이미 배운 사

람에게는요! STL을 안 배운 사람은 p가 두 개의 값을 갖고 있다는 사실을 알기 어렵습니다. 그래서 이 코드는 다음과 같이 쓸 수 있도록 C++17에서 새로운 기능이 추가되었습니다. 고쳐서 전체를 다시 써 보겠습니다.

```
int main( )
{
    vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    auto [min, max] = minmax_element( v.begin( ), v.end( ) );

    cout << "최소: " << *min << ", 최대: " << *max << endl;
}
```

minmax_element 알고리즘이 리턴하는 값이 최소·최대 원소의 **위치**이기 때문에 값을 출력하려면 *을 붙이는 것은 어쩔 수 없습니다. 다수 개의 리턴 값에 내 마음대로 이름을 붙이는 이 기능을 structured binding이라고 합니다. 다른 언어에서는 unpacking이라는 이름을 쓰죠 아마? 함수의 리턴 타입이 **pair**나 **tuple**인 경우에 사용하면 좋겠습니다만 이것도 **syntactic sugar**임을 알아두세요. 앞으로 갑자기 auto [ ] 기호가 나온다고 놀라지 말고 지금 검색해 보고 가세요.

그럼 String의 원소 중에서 최대·최솟값을 찾아볼까요?

```
int main( )
{
    String str { "The quick brown fox jumps over the lazy dog" };

    auto [min, max] = minmax_element( str.begin( ), str.end( ) );

    cout << "최소: " << *min << ", 최대: " << *max << endl;
}
```

지난 시간 강의에서 sort를 해결하지 못했다면 컴파일에 실패할 것입니다. 그래도 관계없습니다. minmax_element 알고리즘이 요구하는 반복자는 forward 반복자이니 큰 문제 없을 겁니다.

오류메시지를 보니 이항연산자 == 로 반복자를 비교하나 봅니다. String_Iterator에 == 연산자를 추가하겠습니다. 이미 != 연산자를 만들었으니 바로 밑에 추가하면 되겠습니다. 사실 지난 시간에 만든 String_Iterator는 forward 반복자의 요구사항을 다 갖추지 못했던 거네요.

```
// String.h에서 class String_Iterator에 추가
```

```
    // 생략

    // 2020. 4. 27 추가
    bool operator==( const String_Iterator& rhs ) const {
            return p == rhs.p;
    }
```

역시 'z'가 가장 큰 값입니다.

minmax_element 함수의 기능을 줄여서 최댓값을 찾는 함수 max_element를 직접 만들어 보겠습니다. 함수 이름 앞에 my_를 붙여주세요. 진짜 max_element가 있거든요.

```
int main( )
{
    String str { "The quick brown fox jumps over the lazy dog" };

    auto p = my_max_element( str.begin( ), str.end( ) );

    cout << "최대: " << *p << endl;
}
```

[실습] 위의 프로그램이 실행되도록 my_max_element를 선언하고 정의하라. (10분)

```
// declaration
template <class T>
T my_max_element( T b, T e );

// definition
template <class T>
T my_max_element( T b, T e )
{
    // 검사할 원소가 없다면
    if ( b == e )
            return e;                    // 마지막 위치를 리턴

    // 제일 처음 원소의 값을 최댓값으로 초기화
    typename iterator_traits<T>::value_type max_value = *b;
```

```
    // 제일 처음 원소의 위치를 최댓값인 위치로 초기화
    T max_position = b;

    while ( b != e ) {
            if ( max_value < *b ) {
                    max_position = b;
                    max_value = *b;
            }
            ++b;
    }
    return max_position;                // 최대값의 위치를 리턴
}
```

생각나는 대로 최댓값을 찾는 코드를 적어본 것입니다. 차차 다듬어 가면 되겠습니다. C++11에서 auto가 도입되기 전에는 최댓값을 위와 같이 반복자의 **value_type**으로 선언해야만 했습니다. 지난 시간에 고생해서 설명한 내용이 조금 쓸모 있어 보이는 순간이었습니다. auto를 사용하여 다시 적겠습니다.

```
    // 제일 처음 원소의 값을 최대값으로 초기화
    auto max_value = *b;
```

그런데 이 함수의 기능을 조금 더 곰곰히 생각해 보면 max_value라는 변수를 선언할 필요도 없음을 알 수 있습니다. 그냥 이렇게 쓰면 됩니다.

```
// 정신 차리고 다시 만들어 본 my_max_element

template <class T>
T my_max_element( T b, T e )
{
    if ( b == e )
            return e;

    T max = b;

    while ( ++b != e )
            if ( *max < *b )
                    max = b;
    return max;
}
```

처음 만든 것 보다 간결합니다. **프로그램은 간결하게 쓸 수 있을수록 좋습니다.** 알고리즘 함수는 반복자를 받고 반복자를 리턴한다는 점을 잊지 마세요. 몇 몇 함수를 제외한 모든 알고리즘 함수는 반복자를 리턴합니다. 최댓값을 직접 넘겨주는 것이 아닙니다. 위치를 리턴합니다.

제네릭 함수 하나를 더 만들어 보겠습니다. 알고리즘 함수 copy입니다. C++ 표준 문서에서 copy의 선언은 다음과 같습니다.

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

copy 함수는 input 반복자로 지정된 [first, last) 범위의 원소를 output 반복자로 지정한 목적지 result로 복사합니다. 목적지는 전체 범위를 지정할 필요가 없습니다. 시작 위치를 지정하면 ++로 다음 위치에 복사할 수 있으니까요.

```
int main( )
{
    char a[ ] { '1', '2', '3', '4', '5' };
    char b[5];

    copy( begin( a ), end( a ), begin( b ) );

    for ( char c : b )
            cout << c << " - ";
    cout << endl;
}
```

실행시켜 보면 a의 모든 원소가 b에 복사된 것을 확인할 수 있습니다.

[실습] 위의 copy를 my_copy로 바꿔 문제없이 실행되도록 my_copy를 선언하고 정의하라. (10분)

```
// declaration
template <class InIter, class OutIter>
OutIter my_copy( InIter first, InIter last, OutIter result );

// definition
template <class InIter, class OutIter>
OutIter my_copy( InIter first, InIter last, OutIter result )
{
    while ( first != last ) {           // 처리해야 할 원소가 있는 동안
            *result = *first;                   // first의 값을 result에 쓴다
            ++result;                           // 다음에 쓸 위치로 이동한다
            ++first;                            // 다음에 읽을 위치로 이동한다
    }
    return result;                      // 이어서 쓸 위치를 리턴한다
}
```

함수의 본문을 잠시 감상해 보세요. 코드에 쓸데없이 설명을 붙이긴 했지만 간결하기 이를 데 없는 코드입니다. 자료의 형식을 따지는 곳은 한 군데도 없습니다. 그저 기능에 집중하는 코드입니다. 제네릭 코드입니다. 이 코드에서 문제가 될 부분은 전달된 어떤 것이 !=, *, ++ 기능을 제공하지 않는다면 이 코드는 컴파일과정에서 실패할 것이라는 것입니다. 이 함수는 반복자라면 모두 갖고 있는 최소 기능만을 요구하기 때문에 어떤 반복자를 전달하더라도 문제될 것이 없습니다.

String을 화면에 복사하는 다음 프로그램을 살펴보겠습니다.

```
int main( )
{
    String str { "The quick brown fox jumps over the lazy dog" };

    my_copy( str.begin( ), str.end( ), ostream_iterator<char>( cout, "*" ) );
}
```

ostream_iterator는 반복자 어댑터 중 하나인 입출력스트림 반복자 중 출력스트림 반복자입니다. 생성자의 인자로 cout을 전달하여 출력을 화면에 합니다. 두 번째 인자는 원소와 원소 사이를 구분할 수 있는 기호입니다. 프로그램을 실행하면 다음과 같이 출력됩니다.

그럼 이번엔 String을 vector로 복사해 보겠습니다.

```
int main( )
{
    String str { "The quick brown fox jumps over the lazy dog" };

    vector<char> v;

    my_copy( str.begin( ), str.end( ), v.begin() );

    for ( char c : v )
            cout << c << '~';
    cout << endl;
}
```

프로그램을 실행시켜 보세요. vector<char> v에 원소들이 잘 복사되었나요?

[실습] 이 프로그램의 실행 결과를 설명하라. (5분)

실행해 보고 각자 왜 이런 결과가 나오는지 스스로 생각해 보라고 일부러 [실습]이라고 붙여 봤습니다. 안된다고 멈추지 말고 이유를 생각해 보는 것이 중요합니다. 실제 강의라면 2~3분 정도 생각해 보라하고 학생들에게 어떻게 생각하는지 물어봤을 텐데 말입니다.

프로그램이 잘못된 곳은 한 군데도 없습니다. 논리적으로 잘못된 것입니다. 복사기를 생각해 보세요. 지금 이 상황은 복사하려고 문서를 가지고 갖는데 A4가 없는 상황입니다. 목적지에 공간이 없어 프로그램이 이상 동작한 것입니다. STL은 반복자가 잘못된 공간을 가리키고 있다는 것을 확인하지 않습니다. 위에서 my_copy 함수는 우리가 직접 만들었습니다. 코드를 다시 한 번 살펴보세요. 목적지인 result가 잘못되었는지 검사하는 부분 비슷한 곳도 없습니다.

STL에서 가장 중요하게 여기는 것은 속도입니다. 그렇게 할 수 있더라도 속도를 느리게 하는 것은 아예 코딩하지 않습니다. 여기에서 생기는 문제점이 있다면 프로그래머가 책임져야 합니다. 그럼 어떻게 해결할 수 있을까요?

공간을 마련해 주면 됩니다. 복사하기에 충분할 만큼 빈 A4를 미리 넣어 놓듯이 말이죠. 이렇게 바꿔 프로그램하면 됩니다.

```
vector<char> v;
v.reserve( str.size( ) );              // 크기에 맞는 공간을 마련하다(복사용지를 넣는다)

my_copy( str.begin( ), str.end( ), v.begin() );
```

어떻습니까? 문제없이 실행되죠?
모두 확인해 주세요.
그럼 넘어갑니다.

네? 안 된 다 고 요?
그럴리가요...

아까와 다르게 프로그램도 정상종료되고 분명히 my_copy 코드를 보면 복사가 안 될 수가 없단 말입니다.

누가 설명을 좀 해 주세요.
분명 틀린 곳은 없는데...

네! 뭐라구요? 잘 안들려요. 조금 큰 소리로 말해 주세요.
... ...
감사합니다. 나보다 똑똑한 학생입니다.
그렇습니다. **분명히 vector에 copy 되었습니다.** 진짜 그런지 먼저 확인해 보겠습니다.

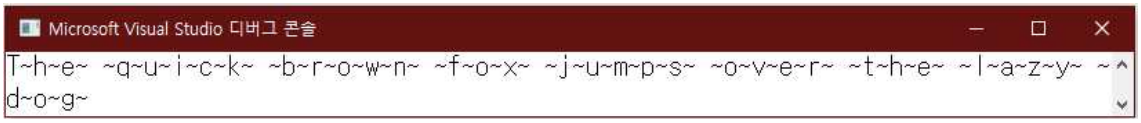
```
int main( )
{
    String str { "The quick brown fox jumps over the lazy dog" };

    vector<char> v;
    v.reserve( str.size( ) );

    my_copy( str.begin( ), str.end( ), v.begin() );

    for ( int i = 0; i < str.size( ); ++i )
            cout << v[i] << '~';
    cout << endl;
}
```

출력하는 for 루프에서 달라진 곳을 찾아보세요. 다음은 출력화면입니다.



모든 원소가 복사되었습니다. 뭐가 잘못되었는지 찾았습니까?

그렇습니다. **알고리즘이 할 수 없는 것이 있습니다.** 바로 vector의 원소 개수를 바꾸는 것입니다. 절대 그렇게 할 수 없습니다. 알고리즘과 컨테이너는 서로 모른다니까요!

그럼 어떻게 컨테이너에 원소를 복사할 수 있겠습니까? 알고리즘 함수의 내용은 전혀 바뀌지 않은 채로 복사하면서, 컨테이너의 원소 수도 제대로 관리할 수 있어야 합니다. 컨테이너의 원소 개수를 바꿀 수 있는 것은 컨테이너 자신 말고는 없습니다. 보통 원소 수가 변경되는 것은 원소가 추가되거나 제거되는 상황입니다. 그렇다면 복사는 원소가 추가되는 상황으로 생각할 수 있습니다. STL의 해결방법은 이렇습니다.

copy 알고리즘의 목적지 인자로 전달되는 반복자를 **"반복자처럼 행동하는 객체"**로 바꿔치기 하는 것입니다. 역시 코드를 보는 것이 쉽습니다.

```
// 코드 7-1
int main( )
{
    String str { "The quick brown fox jumps over the lazy dog" };

    vector<char> v;

    my_copy( str.begin( ), str.end( ), back_inserter( v ) );

    for ( char c : v )
            cout << c << '~';
    cout << endl;
}
```

back_inserter는 반복자처럼 행동하는 객체입니다. 반복자 어댑터의 일종인 **삽입반복자**입니다. 이제 모든 반복자 어댑터를 다 소개했습니다. 뭐가 있었죠? 역방향반복자, 스트림반복자 그리고 지금 소개한 삽입반복자가 있습니다. 참! 이동반복자라는 반복자 어댑터도 있습니다.

다시 반복합니다. my_copy 함수는 달라지지 않습니다. 전달되는 반복자가 다른 것일 뿐입니다. 어떻게 동작하는지 설명할 수 있겠습니까?

```
*result = *first;
```

이 부분이 my_copy에서 값을 쓰는 코드죠. 삽입반복자는 *연산을 삽입동작으로 바꿔 실행하는 것입니다. 이게 비법이죠. 객체 세상에선 어떤 일이 벌어질지 실제 코드를 보기 전까지 알 수 없습니다. 그런데 가능하면 직관적으로 코딩해야하겠습니다. 역방향반복자는 ++ 연산에서 -- 연산을 해도 의미상 아무 문제가 없지만 다른 클래스에서 이렇게 코딩하면 곤란합니다.

back_inserter는 줄여 쓴 것이고 실제 모습은 이렇습니다.

```
my_copy( str.begin( ), str.end( ), back_insert_iterator<vector<char>>(v) );
```

[질문] 코드 7-1에서 고치고 싶은 곳이 있나요?

지금까지 강의를 이해했다면 도전해 보세요. 도전은 진짜 하고 싶은 학생만 하세요. 질문해도 답을 하지 않습니다.

[고급] 다음 프로그램이 의도대로 실행되도록 String_back_insert_iterator(str)를 만들어라. 필요하다면 String 클래스를 변경해도 좋다. (2시간+)

[참고] 이 프로그램을 완성할 수 있다면 컨테이너와 반복자를 밑바닥까지 제대로 이해했다고 스스로 칭찬해도 좋다.

```
int main( )
{
    string s { "The quick brown fox jumps over the lazy dog" };

    String str;

    my_copy( s.begin( ), s.end( ), String_back_insert_iterator(str) );

    cout << str << endl;   // str에는 s의 원소들이 복사되어야 한다.
}
```

컨테이너 어댑터

책 139~195

stack<T>, queue<T>, priority_queue<T>의 기능은 지금까지 공부한 sequence 컨테이너에 비하면 아주 간단합니다. 따라서 이들은 정식 STL 컨테이너에 들어가지는 않습니다. 기존 컨테이너를 이용하여 필요한 기능을 제공할 수 있도록 기존 컨테이너의 인터페이스를 확장하여 이들 클래스를 만들어 사용할 수 있도록 할 뿐입니다. 이렇게 만드는 것을 포장을 바꾼다는 의미로 래핑(wrapping)한다고 부르니 이들 클래스는 넓은 의미로 래핑 클래스라고 불러도 됩니다만 STL에서는 컨테이너 어댑터라고 부릅니다.

컨테이너 어댑터를 빼 놓을 수는 없어 말을 하고 가기는 하지만 내용이 너무 뻔하므로 내 강의에서 설명하지는 않습니다. 컨테이너 어댑터를 꼭 써야 할 순간이 온다면 지금까지 STL을 수강한 여러분은 그저 쓱 한 번 훑어본 후 언제든지 쉽게 사용할 수 있을 것입니다.
틀림없이!

우리 책에서 괜찮은 부분이 컨테이너 어댑터를 다룬 3장에 있어 공부해보라고 소개합니다. 앞부분의 컨테이너 어댑터 설명과 예시는 그냥 넘어가고 175쪽 부터 193쪽까지를 공부하기 바랍니다.

[과제] 알아서 공부하기

곧 진짜 STL 과제를 내겠습니다.