29장. 템플릿과 STL

01_ 템플릿

02_ STL

템플릿 클래스의 사용(1)

- 모든 타입의 배열을 위한 스마트 포인터 클래스

```
template <typename T>
class AutoArray
{
public:
          AutoArray(T* ptr)
          {
                    _ptr = ptr;
          }
          ~AutoArray()
          {
                    delete[] _ptr;
          }
          T& operator[] (int index)
          {
                    return _ptr[index];
          }
private:
          T* _ptr;
};

int main()
{
          AutoArray<float> arr( new float [100] );

          arr[0] = 99.99f;

          return 0;
}
```

템플릿 클래스의 사용(2)

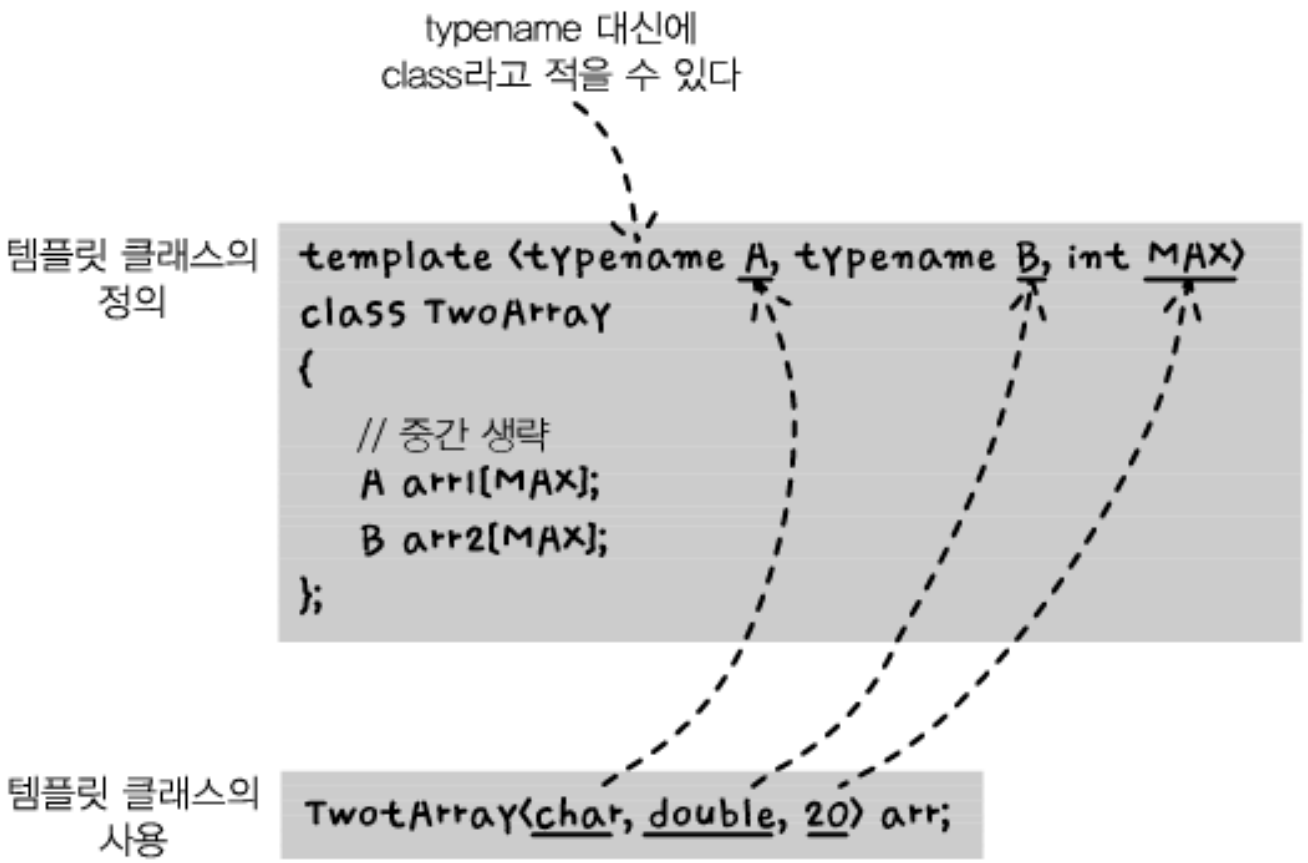
- 템플릿 클래스의 객체를 생성하는 순간에 컴파일러 내부적으로 알맞은 클래스를 만든다.
  - 개발자가 만든 코드

```
template< typename A, typename B, int MAX >
class TwoArray
{
          // 중간 생략
          A arr1[ MAX ];
          B arr2[ MAX ];
};

TwoArray< char, double, 20 > arr;
```

```
class TwoArray_char_double_20          // 이 이름은 임의로 만든 것
{
          // 중간 생략
          char arr1[ 20 ];
          double arr2[ 20 ];
};
```

템플릿 클래스의 사용(3)

- 템플릿 매개 변수의 사용



템플릿 함수의 사용

- 모든 타입을 위한 max() 함수

```
template<typename T>
T max(T a, T b)
{
          return (a > b ? a : b);
}

int main()
{
          int i1 = 5, i2 = 3;
          int i3 = max(i1, i2);                    // i3 = 5

          double d1 = 0.9, d2 = 1.0;
          double d3 = max(d1, d2);       // d3 = 1.0


          return 0;
}
```

템플릿 사용 시 유의할 점

- 템플릿은 컴파일 시간에 코드를 만들어낸다.
  - 그렇기 때문에 템플릿의 사용으로 인한 속도 저하는 생기지 않는다.
  - 하지만, 컴파일 시간이 오래 걸리는 단점이 있다. (하지만 크게 문제가 될 정도는 아님)
- 템플릿 함수의 정의는 헤더 파일에 놓여야 한다.
  - 컴파일 시간에 클래스나 함수를 만들어내기 위해서는 헤더 파일에 위치할 필요가 있다. (컴파일 시간에는 다른 구현 파일의 내용을 확인할 수 없다.)

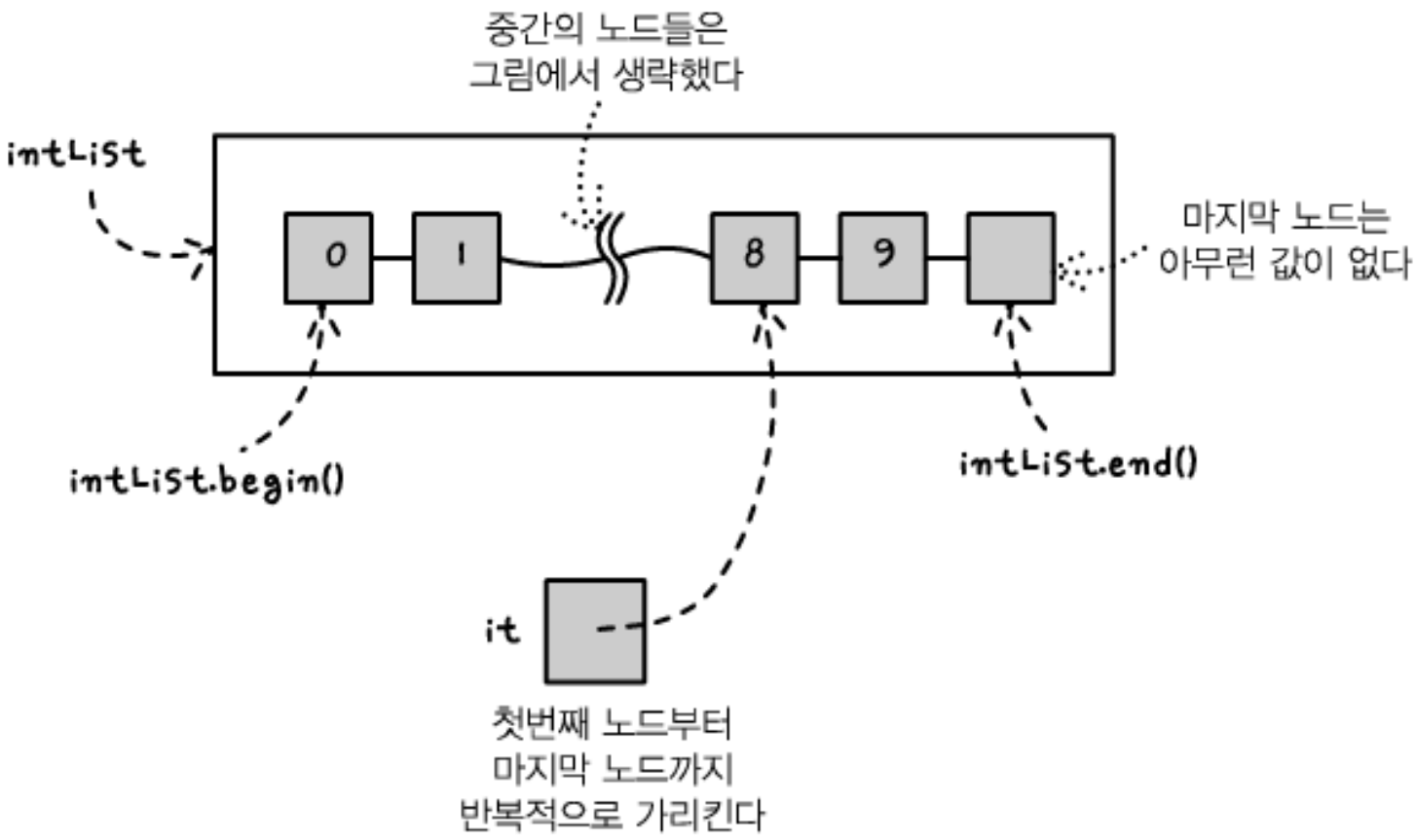
STL 컨테이너(1)

- list 클래스를 사용하는 예

```
#include <list>
#include <iostream>

int main()
{
          // int 타입을 담을링크드 리스트 생성
          std::list<int> intList;

          // 1 ~ 10까지 링크드 리스트에 넣는다.
          for (int i = 0; i < 10; ++i)
                    intList.push_back( i);

          // 5를 찾아서 제거한다.
          intList.remove( 5);

          // 링크드 리스트의 내용을 출력한다.
          std::list<int>::iterator it;
          for (it = intList.begin(); it != intList.end(); ++it)
                    std::cout << *it << "\n";


          return 0;
}
```

STL 컨테이너(2)

- list 클래스의 탐색



STL 컨테이너(3)

- 자주 사용하는 STL의 컨테이너 클래스

클래스	요약
vector	동적인 배열. 동적으로 원소의 개수를 조절할 수 있는 배열이다.
list	링크드 리스트.
deque	배열과 링크드 리스트의 장점을 모아놓은 컨테이너. 배열만큼 원소에 접근하는 시간이 빠른 동시에, 맨 앞과 끝에 원소를 추가하고 제거하는 시간에 링크드 리스트 만큼 빠르다.
map	맵은 원소를 가리키는 인덱스까지도 다양한 타입을 사용할 수 있다. 예를 들어서 다음과 같이 문자열 타입의 인덱스를 사용할 수도 있다. map<string, string> m; m["add"] = "더하다.";

STL 알고리즘(1)

- sort() 함수를 사용하는 예

```
#include <algorithm>
#include <vector>
#include <iostream>

int main()
{
          // 동적 배열을 생성해서 임의의 영문자를 추가한다.
          std::vector<char> vec;
          vec.push_back( 'e');
          vec.push_back( 'b');
          vec.push_back( 'a');
          vec.push_back( 'd');
          vec.push_back( 'c');

          // sort() 함수를 사용해서 정렬한다.
          std::sort( vec.begin(), vec.end() );

          // 정렬 후 상태를 출력한다.
          std::cout << "vector 정렬 후\n";
          std::vector<char>::iterator it;
          for (it = vec.begin(); it != vec.end(); ++it)
                    std::cout << *it;

          return 0;
}
```

STL 알고리즘(1)

- 자주 사용하는 STL의 알고리즘 함수

함수	요약
find()	선형 검색 알고리즘. 선형 검색이란 첫번째 원소부터 하나씩 비교해보는 방법을 말한다.
replace()	특정한 값을 가진 원소를 찾아서 다른 값으로 교체한다.
reverse()	원소들의 순서를 거꾸로 뒤집는다.
sort()	오름차순으로 정렬한다.
binary_search()	이진 탐색 알고리즘. 이전 탐색이란 원소들이 정렬되어 있는 경우에 사용할 수 있는 탐색 방법 중 하나이다.

Q&A