12. 템플릿과 STL

12.1 클래스 템플릿

12.2 STL

12.1 클래스 템플릿

템플릿

- 함수 템플릿, 클래스 템플릿
  - C++의 템플릿을 이용하면 함수나 클래스를 정의할 때 특정 데이터 형을 사용하는 대신 범용형을 사용할 수 있다.
  - 함수 템플릿이나 클래스 템플릿은 여러가지 데이터 형에 대해서 함수 정의나 클래스 정의를 생성할 수 있다.

**템플릿은 처리 알고리즘은 동일하고,
처리할 값의 데이터 형이 다양할 때 유용하게 사용**

템플릿 장점과 제네릭 프로그래밍

- 템플릿 장점
  - 함수 코드의 재사용
    - 높은 소프트웨어의 생산성과 유용성
- 템플릿 단점
  - 포팅에 취약
    - 컴파일러에 따라 지원하지 않을 수 있음
  - 컴파일 오류 메시지 빈약, 디버깅에 많은 어려움
- 제네릭 프로그래밍
  - generic programming
    - 일반화 프로그래밍이라고도 부름
    - 제네릭 함수나 제네릭 클래스를 활용하는 프로그래밍 기법
    - C++에서 STL(Standard Template Library) 제공. 활용
  - 보편화 추세
    - Java, C# 등 많은 언어에서 활용

클래스 템플릿의 선언

- 제네릭 클래스 만들기

```
template <class 범용형이름, ...>
class 클래스이름
{
    // 멤버 변수 정의
    // 멤버 함수 정의
};
```

클래스 템플릿으로 정의한 Stack 클래스

```
template <class T=int>
class Stack {
protected:
    int m_size;
    int m_top;
    T *m_buffer;

public:
    Stack(int size = sz);
   ~Stack();
    void Push(T value);
    T Pop();
    …
};

template <class T>
void Stack<T>::Push(T value) {
                …
}
template <class T> T Stack<T> ::Pop() {
                …
}
…
```

```
// int 타입을 다루는 스택 객체 생성
Stack<int> iStack;

// double 타입을 다루는 스택 객체 생성
Stack<double> dStack;

iStack.Push(3);
int n = iStack.Pop();

dStack.Push(3.5);
double d = dStack.Pop();
```

클래스 템플릿 사용 예

- 개발자가 만든 코드

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

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

- 컴파일러에 의해 생성된 클래스

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

클래스 템플릿의 인스턴스화

- 항상 명시적으로 지정
  - 객체를 생성할 때, 템플릿의 파라미터 지정

```
Stack<int>  s1(10);              // T는 int 형
Stack<string> s2(5);             // T는 string 형
```

  - 객체를 생성하지는 않지만 객체에 대한 포인터나 레퍼런스를 정의하면서 템플릿의 파라미터 지정

```
Stack<char> *pStack = NULL; // T는 char 형
```

  - typedef 문으로 클래스 템플릿의 별명을 정의하면서 템플릿의 파라미터 지정

```
typedef Stack<Point>  PointStack;          // T는 Point 형
PointStack s4(20);                  // Stack<Point> 클래스의 객체 생성
```

클래스 템플릿의 사용

- 어디에서든지 사용 가능

```
void f1(Stack<int>& s);                  // 함수의 인자로 사용

Stack<double>* f2();                     // 함수의 리턴형으로 사용

class X {
protected:
    Stack<string> m_strStack;      // 멤버 객체의 데이터 형으로 사용
    ...
};

class MyStack : public Stack<int> {  ..  };    // 다른 클래스의 기본 클래스로 사용

Template <typename T>
class Array {
  protected:
     T arr[3];
    ..
};
Array< Stack<int> >  arra;                 // 다른 클래스 템플릿의 파라미터로 사용
```

템플릿의 특징

- 실제로 사용되기 전까지는 코드가 생성되지 않음
  - 클래스 템플릿을 인스턴스화 하는 코드를 작성하면 , 컴파일러가 해당 문장을 컴파일할 때 클래스의 코드를 생성함.
  - 컴파일 시간이 오래 걸리는 단점이 있다. (하지만 크게 문제가 될 정도는 아님)

- 코드 크기를 최소화 함
  - 미리 코드를 만들어두는 것이 아니라. 프로그램에 실제로 사용되는 클래스나 함수에 대해서만 코드가 생성됨.

- 함수 템플릿의 정의나, 클래스 템플릿의 멤버 함수의 정의는 헤더 파일에 놓여야 함
  - 컴파일 시간에 클래스나 함수를 만들어내기 위해서는 헤더 파일에 위치할 필요가 있다 (컴파일 시간에는 다른 구현 파일의 내용을 확인할 수 없다).

12.2 STL

STL(Standard Template Library)

- STL = Standard Template Library
  - 표준 템플릿 라이브러리
    - C++ 표준 라이브러리 중 하나
    - 대부분의 C++ 컴파일러는 STL을 지원
  - 성능이 우수하고 안전성이 검증된 라이브러리
  - 많은 템플릿 클래스(제네릭 클래스)와 템플릿 함수(제네릭 함수) 포함
    - 개발자는 이들을 이용하여 쉽게 응용 프로그램 작성
    - 기간도 단축하고 최소한의 노력만으로 원하는 기능을 구현 가능

STL의 장점

- STL의 구성
  - **컨테이너** – 템플릿 클래스
    - 데이터를 담아두는 자료 구조를 표현한 클래스
    - 리스트, 큐, 스택, 맵, 셋, 벡터
  - **iterator** – 컨테이너 원소에 대한 포인터
    - 컨테이너의 원소들을 순회하면서 접근하기 위해 만들어진 컨테이너 원소에 대한 포인터
  - **알고리즘** – 템플릿 함수
    - 컨테이너 원소에 대한 복사, 검색, 삭제, 정렬 등의 기능을 구현한 템플릿 함수
    - 컨테이너의 멤버 함수 아님

STL 컨테이너

컨테이너 클래스	설명	헤더 파일
vector	동적 크기의 배열을 일반화한 클래스	<vector>
deque	앞뒤 모두 입력 가능한 큐 클래스	<deque>
list	빠른 삽입/삭제 가능한 리스트 클래스	<list>
set	정렬된 순서로 값을 저장하는 집합 클래스. 값은 유일	<set>
map	(key, value) 쌍으로 값을 저장하는 맵 클래스	<map>
stack	스택을 일반화한 클래스	<stack>
queue	큐를 일반화한 클래스	<queue>

STL과 관련된 헤더 파일과 이름 공간

- 헤더파일
  - 컨테이너 클래스를 사용하기 위한 헤더 파일
    - 해당 클래스가 선언된 헤더 파일 include

      예) vector 클래스를 사용하려면 **#include <vector>**

      list 클래스를 사용하려면 **#include <list>**
  - 알고리즘 함수를 사용하기 위한 헤더 파일
    - 알고리즘 함수에 상관 없이 **#include <algorithm>**

- 이름 공간
  - STL이 선언된 이름 공간은 std

iterator 사용

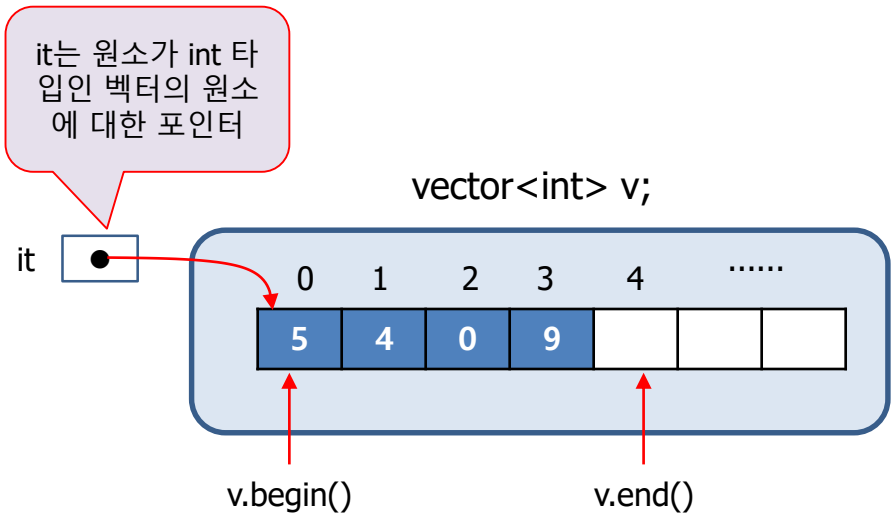
iterator란?

- 반복자라고도 부름
- 컨테이너의 원소를 가리키는 포인터

iterator 변수 선언

- 구체적인 컨테이너를 지정하여 반복자 변수 생성

```
vector<int>::iterator it;
it = v.begin();
```



- begin() 과 end() 함수
  - STL 컨테이너가 공통으로 제공하는 함수
  - begin() 함수 : 첫 번째 원소를 가리키는 iterator를 리턴
  - end() 함수 : 마지막 원소를 가리키는 iterator를 리턴

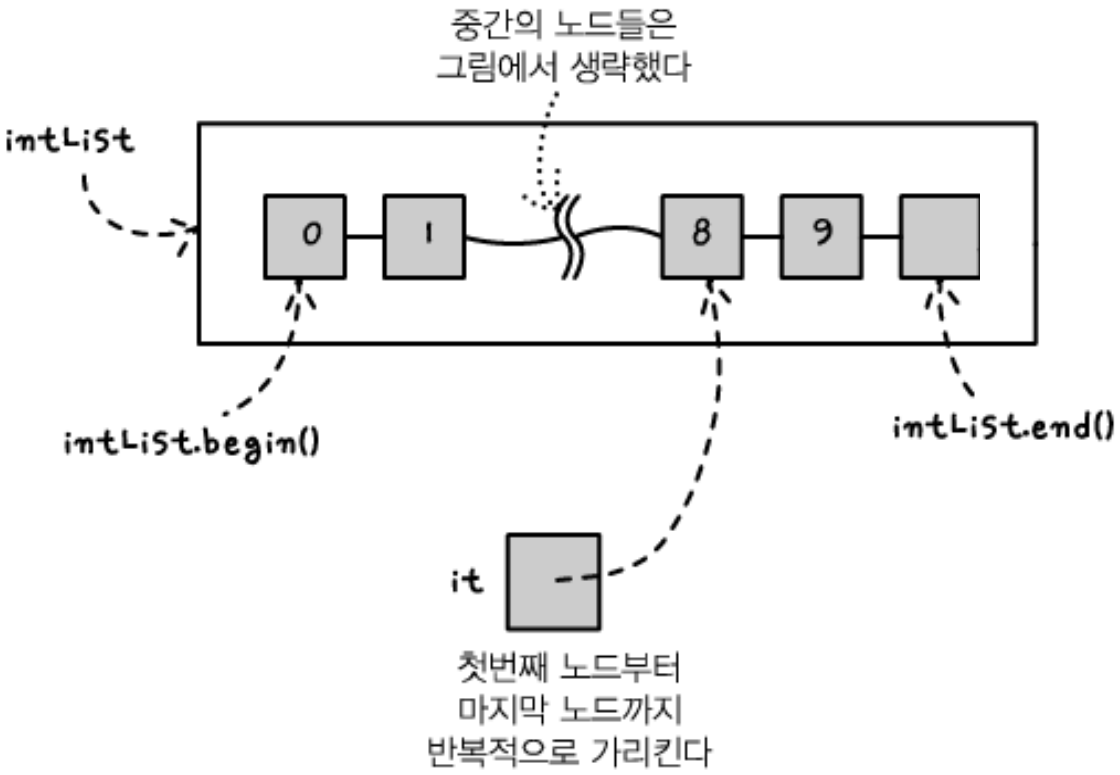
STL의 list 사용하기

Using STLList.cpp

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

int main()
{
	// int 타입을 담을 링크드 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;
}
```

iterator 클래스를 이용한 탐색

```
list<int>::iterator it;       // iterator 클래스 객체 생성
for (it = intList.begin(); it != intList.end();  ++it)
        cout << *it << "\n";
```



STL 컨테이너의 장점

- STL 컨테이너의 종류가 다르더라도 각 컨테이너가 제공하는 인터페이스는 동일
  - list 컨테이너에서 사용했던 push_back, back, begin, end 등의 멤버 함수는 list, vector, deque 등의 다른 STL 컨테이너에서도 동일하게 제공됨.

```
#include <vector>
vector<int> list1;      // vector로 변경해도 프로그램의
                        // 나머지 부분은 그대로 동작한다.
```

- 최소한의 코드 수정으로 컨테이너를 다른 것으로 바꿀 수 있음

  ⇒ 프로그램의 개발이나 유지 보수 시간을 단축하는 데 큰 도움이 됨

STL 알고리즘

- STL이 제공하는 범용 함수
  - 템플릿 함수
  - 전역 함수
    - STL 컨테이너 클래스의 멤버 함수가 아님
  - iterator와 함께 작동

copy	merge	random	rotate
equal	min	remove	search
find	move	replace	sort
max	partition	reverse	swap

STL 알고리즘

- sort() 함수 사례
  - 두 개의 매개 변수
    - 첫 번째 매개 변수 : 소팅을 시작한 원소의 주소
    - 두 번째 매개 변수 : 소팅 범위의 마지막 원소 다음 주소

```
vector<int> v;
...
sort(v.begin(), v.begin()+3); // v.begin()에서 v.begin()+2까지, 처음 3개 원소 정렬
sort(v.begin()+2, v.begin()+5); // 벡터의 3번째 원소에서 v.begin()+4까지, 3개 원소 정렬
sort(v.begin(), v.end()); // 벡터 전체 정렬
```

sort() 함수를 이용한 vector 정렬

STLSort.cpp

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

void main() {
	// 동적 배열을 생성해서 임의의 영문자를 추가한다.
	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.
	sort( vec.begin(), vec.end() );

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

sort() 함수를 이용한 vector 정렬

```
        // 이번에는 배열을 정렬해보자
        // 임의 문자열을 넣은 배열을 만든다

        char arr[5] = {'d', 'c', 'b', 'a', 'e'};

        // sort() 함수를 사용해서 정렬한다.
        sort( arr, arr+5 );

        // 정렬 후 상태를 출력한다.
        cout << "배열 정렬 후\n";
        for (char* p = arr; p != arr + 5; ++p)
                cout << *p;
}
```

제너릭(generic) 프로그래밍

- 제너릭 프로그래밍
  - 정보의 타입과 정보를 처리하는 알고리즘을 분리하여 서로 독립적으로 관리할 수 있는 프로그래밍 방식
    - 컨테이너 : '정보의 타입'
    - sort() 함수를 비롯한 알고리즘 함수 : '정보를 처리하는 알고리즘'

- 제너릭 프로그래밍 목표
  - 정보의 타입과 관계없이 동일한 방법으로 처리할 수 있는 알고리즘을 구현하는 것

제너릭 프로그래밍의 장점

- 타입과 알고리즘간의 불필요한 연관성 제거(decoupling)
  - vector 클래스의 세부 구현을 변경했다고 해도 sort() 함수는 영향을 받지 않음
- 재사용성(reusability) 증가
- 확장의 용이성
  - 공통적인 인터페이스 : begin(), end() 등
  - 새로운 컨테이너 클래스를 만들 때 이 공통의 인터페이스를 유지한다면 STL의 모든 알고리즘을 사용할 수 있다.

정리

- 클래스 템플릿을 정의한다고 해서 바로 클래스가 정의되지는 않는다. 클래스는 템플릿의 파라미터를 지정해야만 생성된다.
- 클래스 템플릿의 파라미터를 지정해서 클래스를 정의하는 것을 클래스 템플릿의 인스턴스화라고 한다. 클래스의 객체를 생성하는 것 외에도 다양한 인스턴스화 방법이 제공된다.
- STL은 표준 C++이 지원하는 템플릿 기반 라이브러리이다.
- STL 컨테이너에는 vector, list, deque, map, set, stack, queue 등이 있다.
- STL 컨테이너는 동일한 인터페이스를 제공하므로 최소한의 노력만으로 STL 컨테이너를 교체할 수 있다.
- STL 알고리즘에는 binary_search, find, for_each, max, min, remove, replace, sort 등이 있다.
- 제너릭 프로그래밍은 데이터를 보관하는 형식과 데이터를 처리하는 알고리즘을 분리할 수 있는 방법을 제공한다.