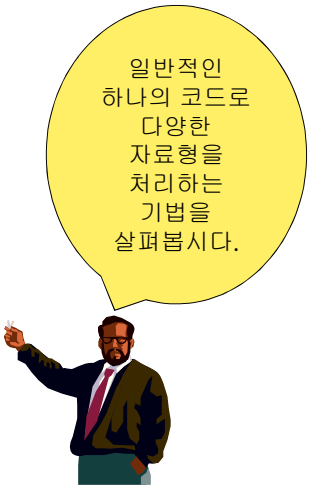
C++ Espresso

제12장 템플릿

이번 장에서 학습할 내용

- 함수 템플릿
- 클래스 템플릿
- 스택 예제



템플릿이란?

- 템플릿(template): 물건을 만들 때 사용되는 틀이나 모형을 의미
  - 붕어빵 틀(template) → 붕어빵 속 재료 다양 가능 → 팥, 슈크림 ...
- 함수 템플릿(function template): 함수를 찍어내기 위한 형틀
- 클래스 템플릿(class template): 클래스를 찍어내기 위한 형틀
- C++ 에서 STL 에서 템플릿 기반 라이브러리 제공 → 14, 15장
- C# 에서는 제네릭(generic, 일반화)

참고) C# 제네릭 메소드

```
class GenericTest  {
    public T1 PrintData<T1, T2>(T1 data1, T2 data2)  {
        T1 ret;   ...
        return ret;
    }
}

class Program  {
    public static void Main()  {
        GenericTest x = new GenericTest();
        int a = x.PrintData<int, string>(5, "안녕하세요");

        string b = x.PrintData<string, int>("안녕하세요", 5);
    }
}
```

함수 get_max() 오버로딩

```
int get_max(int x, int y)
{
    if( x > y ) return x;
    else return y;
}
```

반환값, 매개변수 타입이 다른 여러 함수를 작성

함수 하나만 작성하여(틀을 만들어) 사용하자

```
float get_max(float x, float y)
{
    if( x > y ) return x;
    else return y;
}
```


일반화 프로그래밍

- 일반화 프로그래밍(generic programming): 일반적인 코드를 작성하고 이 코드를 정수나 문자열과 같은 다양한 타입의 객체에 대하여 재사용하는 프로그래밍 기법

int 버전으로 필요하시다구요.

- get_max(1, 2) 로 호출 함수 호출하는 부분 컴파일하면서 실제 함수 본체 생성

템플릿 함수

```
____ get_max(___x , ___ y)
{
  if( x > y)   return x;
  else return y;
}
```

```
int get_max(int  x , int y)
{
  if( x > y)   return x;
  else return y;
}
```

get_max()

T 는 type 대한 변수처럼 사용된다.

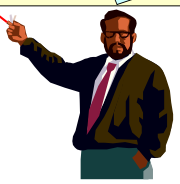
함수 template : T로 전달된 type의 매개변수를 갖는 함수를 만드는 틀

```
template<typename T>
T get_max(T x, T y)
{
    if( x > y ) return x;
    else return y;
}
```

- 다음 둘 다 가능
  template <typename T>
  template <class T>

자료형이 변수처럼 표기되어 있음을 알 수 있다

template<typename T> 작성한 밑의 함수만 적용
→ 함수당 1개 필요



템플릿 함수의 사용

get_max(1, 3) 으로 호출

```
template <typename T>
T get_max(T x, T y)
{
  if(x > y)  return x;
  else return y;
}
```

```
int get_max(int x, int y)
{
  if(x > y)  return x;
  else return y;
}
```

get_max(1.8, 3.7) 으로 호출

```
double get_max(double x, double y)
{
  if(x > y)  return x;
  else return y;
}
```

컴파일러가 전달인자의 자료형을 통해서 호출해야할 함수의 유형을 자동으로 결정해준다.
→ 컴파일러가 전달인자의 자료형에 따라 함수를 생성함.

예제

get_max.cpp

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

template <typename T>
T get_max(T x, T y)
{
  if(x > y)    return x;
  else return y;
}

int main()
{
  // 아래의 문장은 정수 버전 get_max()를 호출한다.
  cout << get_max(1, 3) << endl;

  // 아래의 문장은 실수 버전 get_max()를 호출한다.
  cout << get_max(1.2, 3.9) << endl;

  return 0;
}
```

- 다음과 같이 작성도 가능

```
template <typename T> T get_max(T x, T y) {
    ...
}
```

실행 결과

```
3
3.9
계속하려면 아무 키나 누르십시오 . . .
```

템플릿 함수

- 다음과 같이 type 지정하여 호출 가능

```
template <typename T>
T get_max(T x, T y){
        if (x > y)   return x;
        else return y;
}

int main() {
        cout << get_max<int>(1, 3) << endl;          // int → T
        cout << get_max<double>(1.2, 3.9) << endl;   // double → T
        return 0;
}
```

템플릿 함수의 시그니쳐(원형) 작성

- 템플릿 함수의 시그니쳐(원형) 작성 → 아래와 같이 "template <typename T>" 를 원형과 함수 위에 모두 작성 필요

```
template <typename T>
T get_max(T x, T y);

int main() {
        cout << get_max(1, 3) << endl;
        cout << get_max(1.2, 3.9) << endl;
        return 0;
}

template <typename T>
T get_max(T x, T y){
        if (x > y)   return x;
        else return y;
}
```


템플릿 특수화 기능

- 특별한 매개변수에 대해 다른 동작 하고자 할 때 <> 사용 → 작성 불필요, 일반적인 함수 중복 작성과 동일

```
template <typename T>                    // 함수 템플릿 정의
void print_array(T a[], int n)           // 아래 두 함수 이외의 매개변수인 경우 모두 호출
{                                        // 매개변수 한 개는 T, 한 개는 int 지정
     for(int i=0;i<n; i++)
          cout << a[i] << " ";
     cout << endl;
}

template <>                              // 템플릿 특수화,
void print_array(char a[], int n)        // 매개 변수 char인 경우에는 이 함수가 호출.
{
     cout << a << endl;
}

void print_array(float a[], int n) {
   for (int i = 0; i < n; i++)           // 매개변수가 flaot 인자인 경우
     cout << a[i];
}
```

// **없어도 무관**(아래 함수와 동일하게 함수 중복임.)
// 우선 순위(함수 중복 > 함수 템플릿)

- 각 함수 차이

1st 함수는 원소를 한 칸씩 띠고 출력 (인자는 char, float 아니 경우)
2nd 함수는 빈칸 없이 출력 , (인자 char)
3rd 함수는 빈칸 없이 출력, no endl,(인자 float)

템플릿 함수의 특수화

```
int main()
{
        int a[] = { 1, 2, 3, 4, 5 };
        float b[] = { 11, 12, 13, 14, 15 };
        char c[] = { '1', '2', '3', '4', '5' , '\0'};

        print_array(a, 5);    // 1st 함수 호출
        print_array(b, 5);    // 3rd 함수 호출 , float
        print_array(c, 5);    // 2nd 함수 호출 , char

}
```


함수 템플릿과 함수 중복

```
// 의도 : 인자 2개중  큰 값 반환
template <typename T>
T Max(T a, T b){
        return a > b ? a : b;
}

int main(void){
        char a1[] = “aa";
        char a2[] = “bbb";
        cout << Max(a1, a2) << endl;        // (1)
        return 0;
}
```

- 출력 ??

  aa

- template 로 생성되는 함수는 ?

```
//T → char* 로 받음
char* Max(char* a, char* b){
        return a > b ? a : b;
}
// a, b 저장한 주소 중 큰 주소를 반환
// a 저장 주소가 b 저장 주소보다 큰 모양
```

→ 작성 잘못

함수 템플릿과 함수 중복

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

int main(void){
        cout << Max(11, 15) << endl;        // (1)
        cout << Max(3.5, 7.5) << endl;      // (2)
        cout << Max('T', 'Q') << endl;      // (3)

        char a1[] = "aa";
        char a2[] = "bbb";
        cout << Max(a1, a2) << endl;        // (4)
        return 0;
}
```

```
15
7.5
T
aa
```

- (1), (2) 는 큰 수 구하기 → good
- (3)은 문자 순서 구하기 → 'T' > 'Q'
  - 수, 문자는 뒤에 것이 큼
- (4)는 목적이 문자열 긴 것 구하기인 경우 ??
  - char* a = a1; 으로 주소값 받음
  - a, b 가 저장한 주소값을 비교
  - 결과 보면 a1 주소가 더 큰 모양 ??
- 이와 같이 기존의 템플릿 함수로 원하는 것을 못 구하는 상황에서는
  - 원하는 함수를 중복하여 작성
  - (4)에서 문자열 긴 것 구하기 를 원하는 경우 뒤 코드와 같이 중복 함수 추가해야함

함수 템플릿과 함수 중복

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

int main(void){
        cout << Max(11, 15) << endl;        // (1)
        cout << Max(3.5, 7.5) << endl;      // (2)
        cout << Max('T', 'Q') << endl;      // (3)

        char a1[] = "aa";
        char a2[] = "bbb";
        cout << Max(a1, a2) << endl;        // (4)
        return 0;
}
```

```
// 두 문자열 비교 길이가 긴 것을 반환하는 함수 추가
char * Max(char *a, char *b){
        return strlen(a) > strlen(b) ? a : b;
}
```

- 우선 순위(함수 중복 > 함수 템플릿)
- 앞의 템플릿 특수화와 동일 내용

함수 템플릿과 함수 중복

swap_values.cpp

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

template <typename T>
void swap_values(T& x, T& y)
{
   T  temp;
   temp = x;
   x = y;
   y = temp;
}
void swap_values(char* s1, char* s2)
{
   int  len;

   len = (strlen(s1) >= strlen(s2)) ? strlen(s1) : strlen(s2);
   char* tmp = new char[len + 1];

   strcpy(tmp, s1);
   strcpy(s1, s2);
   strcpy(s2, tmp);
   delete[] tmp;
}
```

템플릿함수

함수 오버로딩 (중복)

```
strcpy_s(tmp, strlen(s1)+1, s1);
strcpy_s(s1, strlen(s2)+1, s2);
strcpy_s(s2, strlen(tmp)+1, tmp);
```

- int, float 등은 템플릿 함수로 처리 가능
- 앞 예제와 같이 템플릿 함수로 안되는 경우 그 경우만 처리하는 함수 작성 필요
→ 문자열 처리 swap 함수 따로 작성
- 우선 순위(함수 중복 > 함수 템플릿)
→ 매개변수가 char* 인 것 먼저 찾아 수행

함수 템플릿과 함수 중복

```
int main()
{
   int x=100, y=200;
   swap_values(x, y);              // x, y가 모두 int 타입- OK!
   cout << x << " " << y << endl;

   char s1[100]="This is a first string";
   char s2[100]="This is a second string";
   swap_values(s1, s2);            // s1, s2가 모두 배열 - 오버로딩 함수 호출
   cout << s1<< " " << s2<< endl;
   return 0;
}
```

두 값을 바꾸는 **swap_values** 함수를 템플릿 하나만 작성하면 배열 전달시 오류 발생 → 배열을 인자로 받는 함수 필요

실행 결과

```
200 100
This is a second string This is a first string
계속하려면 아무 키나 누르십시오 . . .
```

두 개의 타입 매개 변수

```
template<typename T1, typename T2>
void copy(T1 a1[], T2 a2[], int n)
{
    for (int i = 0; i < n; ++i)
        a1[i] = a2[i];
}
```

type 다른 매개변수 처리 가능

```
int main()
{
        int a[] = { 1, 2, 3, 4, 5 };
        float x[5];

        copyy(x, a, 5);      // 함수 이름 copy 로 하면 오류 → 함수이름 고쳐야
}                            // 위의 함수이름도 수정
```


참고)

```
template<typename T>
T max(T& x, T& y)
{
    if(x > y) return x;
    else return y;

}
int main(){
    int i = max(5,6);
    double f = max(6.78, 3.52);
    double g = max(6, 3.52);   // error
}
```

max(6, 3.52); 호출시 오류
T 2개지만 동일 type 필요


중간 점검 문제

1. 변수의 절대값을 구하는 int abs(int x)를 템플릿 함수로 정의하여 보자.
2. 두수의 합을 계산하는 int add(int a, int b)를 템플릿 함수로 구현하여 보자.
3. displayArray()라는 함수는 배열을 매개 변수로 받아서 반복 루프를 사용하여서 배열의 원소를 화면에 출력한다. 어떤 타입의 배열도 처리할 수 있도록 함수 템플릿으로 정의하여 보라.


중간 점검 문제

1) cf) abs 는 이미 정의된 함수 이름

```
template <typename T>
T get_abs(T x) {
	if (x >= 0)   return x;
	else return -x;
}

int main() {
	cout << get_abs(5) << endl;
	cout << get_abs(-5.5) << endl;
	return 0;
}
```

2)

```
template <typename T>
T sum(T x, T y) {
	return x+y;
}

int main() {
	cout << sum(1, 2) << endl;
	cout << sum(5.0, -10.5) << endl;
	return 0;
}
```


중간 점검 문제

3)

```
template <typename T>// 함수 템플릿 정의
void print_array(T a[], int n){
        for (int i = 0; i < n; i++)
                cout << a[i] << " ";
        cout << endl;
}
```

```
int main()
{
        int a[] = { 1, 2, 3, 4, 5 };
        float b[] = { 11, 12, 13, 14, 15 };
        char c[] = { '1', '2', '3', '4', '5' , '\0' };

        print_array(a, 5); // int
        print_array(b, 5);  // float
        print_array(c, 5);  // char

}
```

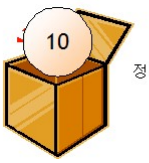

클래스 템플릿

- 클래스 템플릿(class template): 클래스를 찍어내는 틀(template)

```
template <typename 타입이름, ...>
class 클래스이름
{
...

}
```

```
template <typename T>
class Box {
    T data; // T는 타입(type)을
public:
    Box() {       }
    void set(T value) {
        data = value;
    }
```

- 다음 쪽 예제: 하나의 값을 저장하고 있는 박스(클래스 템플릿 사용하지 않은 경우)



정수를 저장하는 상자

예제(클래스 템플릿 사용 하지 않은 경우)

```
class Box {
    int data;
public:
    Box() { }
    void set(int value) {
        data = value;
    }
    int get() {
        return data;
    }
};

int main()
{
   Box box;
   box.set(100);
   cout << box.get() << endl;
   return 0;
}
```

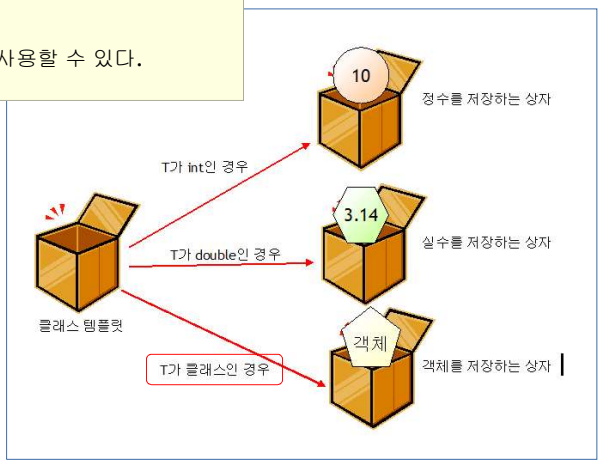
실행 결과

```
100
계속하려면 아무 키나 누르십시오 . . .
```

```
// float 멤버 가진 클래스 필요하면 따로 작성해야 함.
// 거의 비슷한 코드 중복

class Box1 {
        float data;
public:
        Box1();
        void set(float value);
        float get();
}
```


클래스 템플릿 버전

```
template <typename T>
class 클래스이름
{
...// T를 어디서든지 사용할 수 있다.
}
```



예제(클래스 내부에 함수 정의)

클래스 템플릿

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

template <typename T>
class Box {
    T data; // T는 타입(type)을 나타낸다.
public:
    Box() {      }
    void set(T value) {
        data = value;
    }
    T get() {
        return data;
    }
};
```

기존 정수 저장 클래스

```
class Box {
    int data;
public:
    Box() { }
    void set(int value) {
        data = value;
    }
    int get() {
        return data;
    }
};
```


477 쪽 예제

- 클래스 템플릿 사용시 "class<type> 객체" 형식으로 객체 생성

```
int main() {
        Box<int> box;                    // 객체 생성시 type 지정
        box.set(100);
        cout << box.get() <<endl;        // 클래스_이름<type_매개변수> 객체명

        Box<double> box1;
        box1.set(3.141592);
        cout << box1.get() <<endl;

    return 0;
}
```

```
template <typename T>
class Box {
    T data; // T는 타입(type)을 나타낸다.
public:
    Box() {      }
    void set(T value) {
        data = value;
    }
    T get() {
        return data;
    }
};
```

실행 결과

```
100
3.14159
계속하려면 아무 키나 누르십시오 . . .
```

클래스 외부에 템플릿 함수정의

```
template <typename T>
class Box {
    T data; // T는 타입(type)을 나타낸다.
public:
    Box();
    void set(T value);
    T get();
};

template <typename T>
Box<T>::Box() {
}

template <typename T>
void Box<T>::set(T value) {
  data = value;
}

template <typename T>
T Box<T>::get() {
  return data;
}
```

템플릿 멤버함수 외부선언시 형식

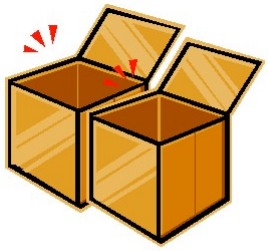
- 멤버 함수마다 template<typename T> 작성
- 모든 멤버 함수 이름에 "클래스이름<T>::함수이름" 지정
  - Box<T>::함수이름

외부선언시 템플릿 멤버함수 형식

반환형 클래스이름<T>::함수이름(인자목록)

두 종류의 타입 매개 변수

- 두 종류의 데이터를 저장하는 클래스 **Box2**



Box2 클래스 템플릿

예 제

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

template <typename T1, typename T2>
class Box2 {

  T1 first_data; // T1은 타입(type)을 나타낸다.
  T2 second_data; // T2는 타입(type)을 나타낸다.

public:
  Box2() {      }
  T1 get_first();
  T2 get_second();
  void set_first(T1 value) {
       first_data = value;
  }
  void set_second(T2 value) {
       second_data = value;
  }
};
```

두개의 타입 매개 변수를 가지는 클래스 템플릿

예 제

```
template <typename T1, typename T2>
T1 Box2<T1, T2>::get_first() {
  return first_data;
}
template <typename T1, typename T2>
T2 Box2<T1, T2>::get_second() {
  return second_data;
}
int main()
{
  Box2<int, double> b;
  b.set_first(10);
  b.set_second(3.14);
  cout << "(" << b.get_first() << ", " << b.get_second() << ")" << endl;
  return 0;
}
```

- get_first() 멤버함수에서는 T1 만 필요하지만
- 클래스 정의에서 사용된 T1, T2 2개 모두 작성해야함 → T1 Box2<T1, T2>::함수이름

실행 결과

```
(10, 3.14)
계속하려면 아무 키나 누르십시오 . . .
```

중간 점검 문제

1. 클래스 템플릿 형태로 라이브러리를 제공하면 어떤 장점이 있는가?

2. 세개의 데이터를 가지고 있는 Triple라는 클래스를 클래스 템플릿으로 작성하여 보라.

템플릿 매개변수에는 변수의 선언이 올 수 있습니다.

```
template <typename T, int len>
class SimpleArray
{
private:
    T arr[len];
public:
    T& operator[] (int idx)
    {
        return arr[idx];
    }
};
```

SimpleArray<int, 5> i5arr;

```
class SimpleArray<int, 5>
{
private:
    int arr[5];
public:
    int& operator[] (int idx) { return arr[idx]; }
};
```

SimpleArray<double, 7> i7arr;

```
class SimpleArray<double, 7>
{
private:
    double arr[7];
public:
    double& operator[] (int idx) { return arr[idx]; }
};
```

템플릿의 인자로 변수의 선언이 올 수도 있다!

템플릿 인자를 통해서 SimpleArray<int, 5>와 SimpleArray<int ,7>이 서로 다른 자료형으로 인식되게 할 수 있다.

함수 템플릿 사용시 파일 분할

● 하나의 파일에 모든 내용을 작성하는 경우 → **아래 / 옆의 코드(함수 위/아래 작성) 문제 없음**

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

template <typename T>
T sum(T a, T b) {
   return (a + b);
}

int main(void){
   int sum_i;
   double sum_d;

   sum_i = sum(2, 3);

   sum_d = sum(2.2, 3.3);

   cout << "int sum : " << sum_i << "....  double sum : "
        << sum_d << endl;

   return 0;
}
```

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

template <typename T>
T sum(T a, T b);

int main(void){
   int sum_i;
   double sum_d;

   sum_i = sum(2, 3);

   sum_d = sum(2.2, 3.3);

   cout << "int sum : " << sum_i
        << "....  double sum : " << sum_d << endl;

   return 0;
}

template <typename T>
T sum(T a, T b) {
   return (a + b);
}
```


함수 템플릿 사용시 파일 분할 – Error

● 소스파일 2, 헤더파일 1개로 분할하여 작성 : 함수 본체는 소스 파일, 시그니쳐(함수 원형)은 헤더파일에 작성

main.cpp

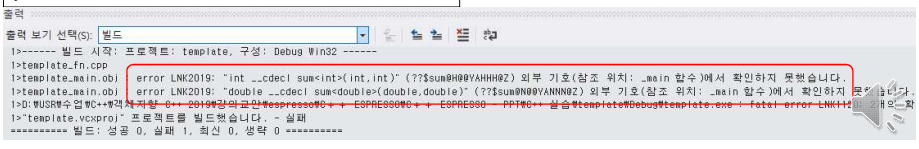
```
#include <iostream>
#include "signature.h"

using namespace std;

int main(void){
   int sum_i;
   double sum_d;

   sum_i = sum(2, 3);

   sum_d = sum(2.2, 3.3);

   cout << "int sum : " << sum_i << "....  double sum : "
      << sum_d << endl;

   return 0;
}
```

fn.cpp

```
#include <iostream>

template <typename T>
T sum(T a, T b) {
      return (a + b);
}
```

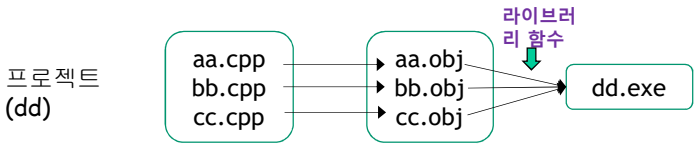
signature.h

```
#pragma once

template <typename T>
T sum(T a, T b);
```



프로그램 컴파일 실행 과정

- 컴파일(Compile)
  - 소스 프로그램에 문법적 오류가 있는지 검사
  - 기계어로 번역, 목적 파일(~.obj) 생성
- 링킹(Linking)
  - 여러 목적 파일을 묶어 하나의 실행 파일(~.exe)로 만들어 주는 기능
- 실행파일(exe 파일) 실행



참고) 파일 분할시 일반적 컴파일 과정(템플릿 함수 없는 경우)

main.cpp

```
#include <iostream>
#include "signature.h"

using namespace std;

int main(void){
   int sum_i;
   double sum_d;

   sum_i = sum(2, 3);

   cout << "int sum : " << sum_i << endl;

   return 0;
}
```

fn.cpp

```
#include <iostream>

int sum( int a, int b) {
     return (a + b);
}
```

signature.h

```
#pragma once

int sum(int a, int b);
```

- 컴파일 → 각 문장의 문법을 검사하는 과정, 컴파일러가 C++ 문법에 따라 내용을 검사하여 목적파일(obj 파일) 생성
  - main.cpp → 컴파일러가 sum(…) 함수를 만나면 위에서 include 한 (signature.h) 컴파일하여 아는 내용, 본체는 다른 파일에 있다는 의미 → 나중 링크시 연결하면 됨. → 컴파일 ok
  - fn.cpp 컴파일 → 문법적으로 문제 없기에 → 컴파일 ok

- 링크 : 모든 obj 파일에서 main.cpp 컴파일 시 연결 못한 함수 sum(…) 함수 찾음 → fn.obj 에 있음 , 연결 Ok

함수 템플릿 사용시 파일 분할(한 파일에 작성시 컴파일 과정)

● 하나의 파일에 모든 내용을 작성하는 경우 → 아래 코드 문제 없음

```
#include <iostream>

using namespace std;

template <typename T>
T sum(T a, T b) {
   return (a + b);
}

int main(void){
   int sum_i;
   double sum_d;

   sum_i = sum(2, 3);

   sum_d = sum(2.2, 3.3);

   cout << "int sum : " << sum_i << "....  double sum : "
        << sum_d << endl;

   return 0;
}
```

```
int sum(int a, int b) {
   return (a + b);
}


double sum(double a, double b) {
   return (a + b);
}
```

● main.cpp → 컴파일 할때 sum(…) 문장을 만나면 template 정의에 따라 함수 2개를 각각 생성, 연결

- 참고) 컴파일시 함수 연결 : 함수 호출한 문장과 해당 함수가 있는 메모리 주소를 연결

함수 템플릿 사용시 파일 분할 (컴파일 과정) LNK Error

main.cpp

```
#include <iostream>
#include "signature.h"

using namespace std;

int main(void){
   int sum_i;
   double sum_d;

   sum_i = sum(2, 3);  // (1)

   sum_d = sum(2.2, 3.3);   // (2)

   cout << "int sum : " << sum_i << "....  double sum : "
      << sum_d << endl;

   return 0;
}
```

fn.cpp

```
#include <iostream>

template <typename T>
T sum(T a, T b) {
      return (a + b);
}
```

signature.h

```
#pragma once

template <typename T>
T sum(T a, T b);
```

● 컴파일
  - main.cpp 컴파일 시 → (1), (2) 문장 컴파일 할 때 헤더파일에 정의가 있으니 본체는 어딘가 있겠네… → 컴파일은 ok → but, template 정의가 없기에 함수 본체 생성 못함(무엇을 하는 함수인지 모르기 때문에)
  - fn.cpp 컴파일 → 문법적으로 문제 없기에 → 컴파일 ok

● 링크 : 모든 obj 파일에서 main.cpp 컴파일 시 연결 못한 함수 sum(…) 2개 함수 찾음 → 없음, 못 찾음 → error

함수 템플릿 사용시 파일 분할 → 해결책 1(일반적)

main.cpp

39쪽 코드와 같이 구성되도록 하면 OK

```
#include <iostream>
#include "signature.h"

using namespace std;

int main(void){
   int sum_i;
   double sum_d;

   sum_i = sum(2, 3);  // (1)

   sum_d = sum(2.2, 3.3);  // (2)

   cout << "int sum : " << sum_i << "....  double sum : "
      << sum_d << endl;

return 0;
}
```

signature.h(or signature.hpp)

```
#pragma once

template <typename T>
T sum(T a, T b) {
        return (a + b);
}
```

- 일반 함수는 정의가 아니라 구현 코드
- 템플릿은 정의에 해당 : 여러 함수들의 틀(템플릿)을 정의한 것
- 템플릿은 정의를 작성하는 헤더파일에 작성하는 것이 일반적

● 함수 템플릿을 헤더파일에 작성하고 이를 사용하는 소스 파일에서 포함(include) 한다.
● include 하면 하나의 파일(소스 파일)에 모두 있는 상태(39쪽 코드와 동일)

- 컴파일 : main.cpp 컴파일 시 (1), (2) 문장 컴파일 할 때 헤더파일의 함수 템플릿에 따라 함수 본체를 2개 생성, 연결 → 컴파일 ok
- 링크 : 하는일 없음, ok

클래스 템플릿 사용시 파일 분할 → 해결책(일반적)

aa.h

```
#pragma once

#include <iostream>
using namespace std;

template <typename T>
class Point
{
private:
   T xpos, ypos;
public:
   Point(T x = 0, T y = 0);
   void ShowPosition() const;
};

template <typename T>
Point<T>::Point(T x, T y) : xpos(x),
ypos(y) {  }

template <typename T>
void Point<T>::ShowPosition() const
{
   cout << '[' << xpos << ", " << ypos << ']'
      << endl;
}
```

main.cpp

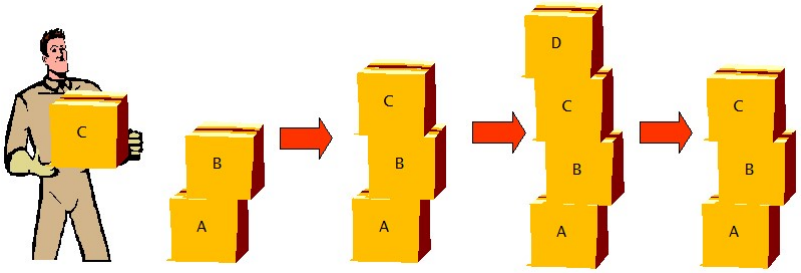
```
#include <iostream>
#include "aa.h"

using namespace std;
int main(void)
{
    Point<int> pos1(3, 4);
    pos1.ShowPosition();
    Point<double> pos2(2.4, 3.6);
    pos2.ShowPosition();
    Point<char> pos3('P', 'F');
    pos3.ShowPosition();
    return 0;
}
```

main.cpp 소스파일을 컴파일 할 때 Point<int>, Point<double> 템플릿 클래스가 만들어져야 한다. 따라서 Point 클래스 템플릿의 멤버함수 정의에 대한 정보도 필요하다.

- 해결책 → 헤더파일에 클래스 템플릿의 모든 내용을 포함시킨다.
- "~~.hpp" 파일에 작성 가능하고 include 해도 가능

예제: 스택

- 스택(stack): 후입 선출(LIFO:Last-In First-Out) 자료 구조



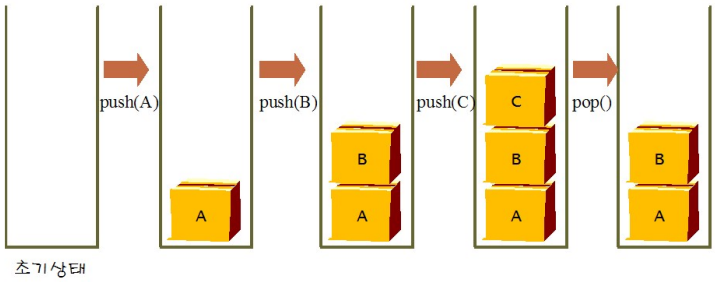
template class 로 stack 구현

int top, size
저장공간

스택의 연산들

s는 특정 stack 의미, 인자 아님

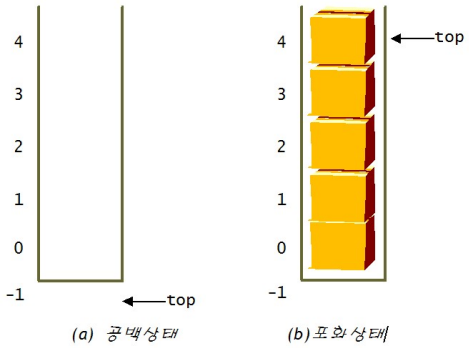
- is_empty(s) ::= 스택이 비어있는지를 검사한다.
- is_full(s) ::= 스택이 가득 찼는가를 검사한다.
- push(s, e) ::= 스택의 맨 위에 요소 e를 추가한다.
- pop(s) ::= 스택의 맨 위에 있는 요소를 삭제한다.



스택의 공백 상태와 포화 상태

top (7장과 다르게 구현)
- 가장 나중에 저장된 데이터 index
- 스택 빈 경우 -1(초기값)
- 한 개 저장하면 top++ 하고 top 에 저장
- 4개 저장시 값 4

cf) 배열 5개
index (0~4)



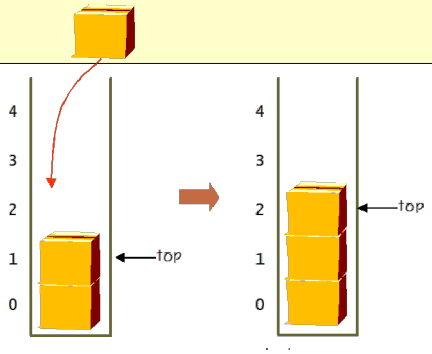
isEmpty() , isFull()

```
isEmpty()

if top = -1
    then return TRUE
    else return FALSE
```

```
isFull()

if top = (MAX_STACK_SIZE-1)
    then return TRUE
    else return FALSE
```

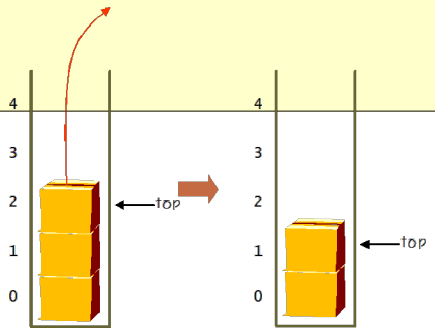
push()

```
push(x)

if isFull()
    then error "overflow"
    else top←top+1
         stack[top]←x
```



pop()

```
pop(x)

if isEmpty()
    then error "underflow"
    else e←stack[top]
         top←top-1
         return e
```



스택의 구현

```
int main() {
    Stack<int> s;  // 크기가 100인 정수형 스택, 클래스 템플릿으로 구현
    try{
        s.push(100);        s.push(200);        s.push(300);        s.push(400);
        cout << s.pop() << endl;                cout << s.pop() << endl;
        cout << s.pop() << endl;                cout << s.pop() << endl;
        cout << s.pop() << endl;
    }
    catch (FullStack e) {          // push 함수에서 full 경우 error 객체 던짐, FullStack 클래스 구현
        e.Show();
    }
    catch (EmptyStack e) {         // pop 함수에서 empty 경우 error 객체 던짐, EmptyStack 클래스 구현
        e.Show();
    }
    return 0;
}
```

```
400
300
200
100
stack empty -> stack size :100  top :-1 ← EmptyStack 클래스 멤버함수에서 출력
```


스택의 구현

```
#include <iostream>
using namespace std;
// 예외 처리를 위한 클래스
class FullStack {
        int top, size;          // 스택의 top, 스택 크기 저장
public:
        FullStack(int t, int s) : top(t), size(s) { }
        void Show() {
                cout << "stack full -> stack size :" << size << "  top :" << top << endl;
        }
};
// 예외 처리를 위한 클래스
class EmptyStack {
        int top, size;          // 스택의 top, 스택 크기 저장
public:
        EmptyStack(int t, int s) : top(t), size(s) { }
        void Show() {
                cout << "stack empty -> stack size :" << size << "  top :" << top << endl;
        }
};
```

예외(full, empty) 발생시
예외객체 생성 전달

스택의 구현

```
template < typename T>
class Stack {
private:
     T* s;          // 저장 공간, 임의 type 데이터 저장 가능
     int   size, top;
public:
     Stack(int n = 100) : size(n), top(-1)  { // 기본 크기 100
          s = new T[size];
     }
     ~Stack() {     delete []s;    }
     void  push(T v);
     T pop();
     bool  isEmpty() const {      return top == -1;       }
     bool  isFull() const {       return top == size - 1; }
};
```


스택의 구현

```
template< typename T >
void Stack< T >::push( T v ) {
  if ( isFull() )
          throw FullStack(top, size);          // 예외객체 생성하여 던짐
  s[ ++top ] = v;
}

template< typename T >
T Stack< T >::pop( ) {
  if ( isEmpty() )
          throw EmptyStack(top, size);        // 예외객체 생성하여 던짐
  return s[ top-- ];                  // s[top] 반환하고 top 감소
}
```