**템플릿 (template)**

템플릿을 사용하여 프로그래밍을 하는 것을 **일반화(Generic - 제네릭)** 프로그래밍이라고 한다.

템플릿을 알아야 후에 **STL (Standeard Template Library)** 를 공부 할 수 있다.

템플릿은 붕어빵 모형틀, 모형 자 등등으로 비유 할 수 있으며, **함수나 클래스의 모양을**

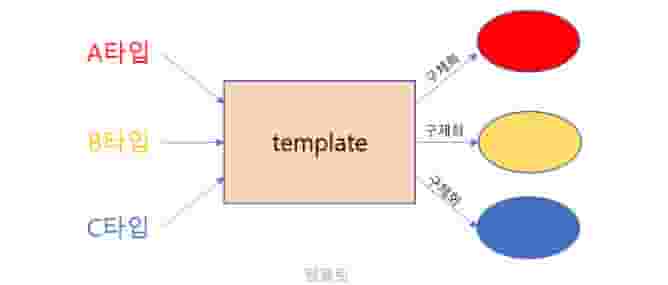
**만들 수 있는 틀**이다. 예를 들어, 함수나 클래스가 붕어빵이라면 템플릿은 그것을 만들기 위한

붕어빵 틀인 셈이다.

**일반화(Generic)** 이라는 말의 의미는 붕어빵 틀을 만드는 것과 같다.

만약 이러한 틀이 없다면 붕어빵 모양을 손수 빗어서 만들어야 하지만 이런 특정 기능을 하는

틀을 만들어 놓는다면 **어떤 재료를 넣든** **결과물은 붕어빵의 모양**일 것이다.

****

위 그림과 같이 찍어내는 모양은 같지만 타입에 따라서 색은 달라진다. 즉, 내용에 따라 객체의 타입이 달라진다는 것이다. 공통점은 같은 틀, 템플릿에 찍힌 모양일 뿐, 내용은 같아지지 않는다.

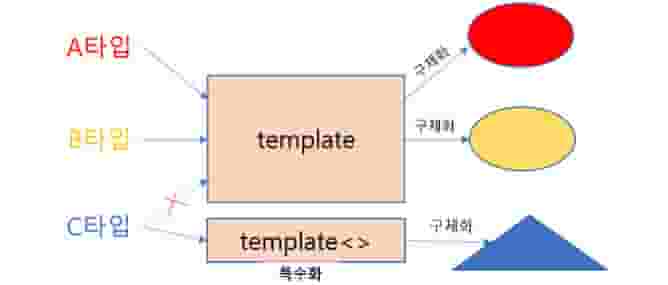
위 구조는 각 타입에 대한 함수를 오버 로딩하여 만들 수도 있지만, 이처럼 템플릿을 사용한다면

**하나의 템플릿만으로 각 타입의 함수를 오버 로딩하는 효과**를 가질 수 있다.

그렇게 된다면 코딩이 좀 더 심플해지고, 작업을 훨씬 효율적으로 수행 할 수 있다.

이것이 **일반화 프로그래밍의 목표**다.

위 그림처럼 코딩을 할 때, **특정 타입의 모양은 붕어빵이 아니라, 다른 빵의 모양 또는 다른 기능을 수행**하는 상황이 충분히 일어 날 수 있다.



그 상황은 위의 그림으로 이해 할 수 있었다. 이것을 **템플릿 특수화**라고 한다.

템플릿 특수화도 함수의 오버 로딩과 비슷하게 이해 할 수 있고, 복잡하게 생각 할 필요 없이

**특수화를 하려는 타입에 대한 템플릿을 하나 더 정의**하면 해결된다.

위 그림이 적절한 예시이다.

클래스도 그렇듯 템플릿 역시 기본적인 문법이 존재한다.

우선 너무나도 당연하지만 만들기 위해선 선언이 필요하고, 선언 방법은 밑과 같다.

**template <typename N> // 템플릿 선언**

N 이라는 타입에 대해 템플릿을 선언한다는 뜻이다. 만일 여러 타입에 대한 템플릿을 만들고자 한다면 **template<typename N, typname N1, .. >** 와 같이 선언하면 된다.

여기서 N 은 모든 타입을 대변하는 이름이다. 사용자가 해당 템플릿을 사용하려면 N 타입으로 선언된 위치에 값이나 변수를 넣어서 템플릿을 호출하면 대입된 타입이 컴파일 시간에 자동으로 타입이 추론되어 해당 타입에 맞는 함수 또는 클래스가 구체화 된다.

**N myFunc(T a, T b) { return a + b; }**

라는 간단한 템플릿을 만든다면 **내용이 무엇이든지 구체화 되는 모양은 a + b** 가 될 것이다.

만약 **myFunc(1,3)** 이라고 쓰게 된다면 **int myFuns(int, int)** 가 구체화 될 것이고

**myFunc(1.3,3.3)** 이라고 쓰면 **double myFuns(double,double)** 가 구체화 될 것이다.

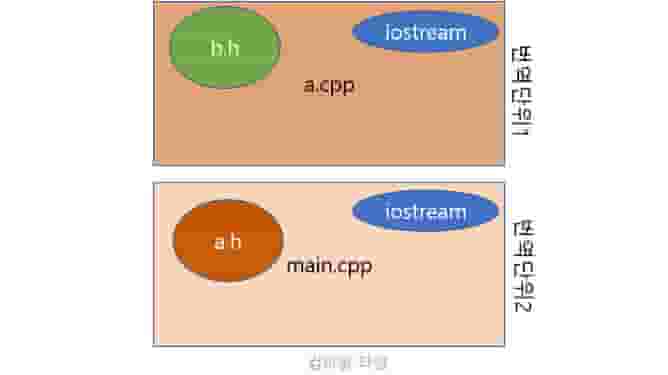
템플릿의 구체화는 컴파일 할 때, 필요한 타입에 대해서만 구체화 해준다.

이렇듯 코드로 보면 확실히 느껴지는 게, **본래는 함수 오버 로딩을 두 번 해야 할 일을**

**템플릿 하나로 간단하게 처리**하는 모습을 볼 수 있다.

이제 템플릿을 구체화 할 때 생각해야 될 문제가 있는데, C++ 은 헤더파일과 cpp 파일을 분리해서 함수, 클래스 선언과 정의를 따로 하게 되는데 일반적으로 **헤더파일에는 선언만 하지 정의는 하지 않는다.**.고 한다. 나는 처음 안 사실이지만 그렇다고 한다.

이 때, 이런 방법으로 **선언과 정의를 따로 하면 템플릿은 문제**를 발생시킨다.



위 그림은 컴파일 타임에 소스가 번역되는 단위를 간략하게 보여주는 예시다.

번역단위1 과 번역단위2 는 컴파일 때 **서로를 참조하지 않는다.**

각각 따로 컴파일이 되고 목적파일로 된다.

보통은 a.h 에는 a.cpp 에서 정의된 함수나 클래스에 대한 선언을 하기 때문에 컴파일 타임에

아무 문제도 발생하지 않는다고 한다. 구체적인 구현에 대한 **연결은 링크 과정에서 링커**에게

맡기게 되어 있기 때문이다.

하지만 템플릿은 구체화라는 작업을 **컴파일 타임**에 하기 때문에 구체적인 구현이 다른 변역단위에 있다면 문제가 발생한다. 즉, 템플릿의 구체화를 위해서는 **템플릿 정의가 같은 번역 단위에 있어야 비로소 구체화가 가능하다.**

즉, 문제가 발생하는 경우는 일반적으로 헤더와 구현을 따로 하듯이 **템플릿을 헤더에 선언하고, 구현부에 정의**한다면 이런 방식은 구체화가 되는 시점에서 컴파일 에러가 발생한다.

결국 템플릿은 어쩔 수 없이 **헤더파일에 선언과 정의가 같이 있어야 한다.**

아마도 둘은 작업하는 시점이 달라서 발생하는 에러 인 것 같다.

즉, 요점은 템플릿은 **헤더파일에서 시작해서 헤더파일에서 끝나야 한다** 인 것 같다.

그냥 간단하게 생각해서 **헤더에서 선언 한 후, 바로 정의** 해버리자.

그리고 템플릿은 컴파일 타임에 템플릿 함수가 호출 될 때 자동으로 이루어지는데, **전달된 타입에 대한 추론**은 컴파일이 알아서 해준다만, 몇 가지의 경우에는 구체화 될 함수의 타입을 강제로 지정해줘야 하는 경우가 있다.

예로 myFunc(1.4, 8) 와 같이 함수를 썼다고 하자.

1.4 와 8 이 서로 다른 타입이라고 볼 수 있을까? 일반적으로 변수를 선언할 때 double x = 8; 이라고 하는 것은 에러가 아니기 때문에 암묵적으로 8 이라는 정수가 double 형으로 **형변환**이 될 수 있기 때문이다.

그런데 **템플릿은 이러한 암묵적인 변환은 허용하지 않는다**고 한다.

변환되지 않고 그대로 해석된다는 말이다. (double 이면 double, int 면 int)

다행히 이런 경우를 위해 사용자가 직접 타입을 강제하는 문법이 있었다.

myFunc<double> (1.4 , 8) 로 한다면 8 의 타입은 double 로 강제되어 결과값은 9.4 가 되고,

myFunc<int> (1.4 , 8) 가 된다면 1.4 의 타입이 int 로 강제되어 결과값은 9 가 된다.

즉, 컴파일러에서 **추론하기 전에 미리 해당 인수에 대한 타입을 강제**하는 것이다. 이 경우엔 타입 추론 대신 해당 데이터를 형변환 시킨다.

**반환 타입에 대한 추론**은 컴파일러가 할 수 없다고 한다.

만약 템플릿이 **인수와 반환 타입을 각각 따로** 다루게 된다면 코드상에서는 반환 타입을 알 수 없으므로 곤란하게 된다. ~~모호성의 문제인가?~~

보통 이런 경우는 반환 타입을 명시적으로 지정하는 목적으로 설계가 된다.

예로 타입을 캐스팅 하는 함수를 만든다면 **반환 타입을 직접 지정해주어야 한다.**

그리고 이제 아마도 우리가 계속해서 쓰게 될 **클래스 템플릿**, 아마 템플릿을 함수로 선언하기보단, **클래스 템플릿을 사용하는 경우가 훨씬 많을 것 같다.**

**함수와 클래스의 구조적 차이점으로 인해 클래스 템플릿은 반드시 타입을 명시**해주어야 하는 점을 빼곤 크게 다른 점은 없다고 한다.

객체가 생성 되기 전(=생성자 호출 전)에 먼저 객체를 위한 메모리가 할당되어야 하는데 메모리 할당을 위해서는 생성자에 전달 될 인수의 타입을 미리 알아야 한다.

이제 와서 멤버 변수의 타입도 모르는데 메모리가 얼마나 할당 되는 지 모르는 건 상식이다.

이렇듯 클래스의 생성자가 호출되기 전에 객체를 위한 메모리가 할당되어 있어야 한다.

만일 클래스 템플릿을 호출할 때, MyClass(3) 과 같이 명시적으로 타입을 지정하지 않고 생성자를 호출하면 **얘가 어떤 타입인 지 모르기 때문에 얼마나 메모리를 할당해야 할 지 모르게** 되고 그로 인해 문제가 발생하는 듯 하다.

이 뿐만이 아니라 템플릿 클래스의 이름이 본래 **MyClass<T>** 라면, int 타입으로 명시적으로

구체화 한 후의 클래스의 이름은 **MyClass<int>** 가 된다고 한다.

이 부분은 함수로 만드는 것과 확실히 구별해서 기억하는 편이 좋을 것 같다.