

# 15장 템플릿

# 강의 목표

- 템플릿 사용의 동기와 장점 이해 (15.2)
- 형식 매개변수(type parameter)를 가지는 템플릿 함수 선언 (15.2)
- 템플릿을 사용하여 일반화 정렬 함수 개발 (15.3)
- 클래스 템플릿을 이용한 일반화 클래스 개발 (15.4-15.5)

# 템플릿의 기본

템플릿의 필요성을 설명하기 위하여 간단한 예를 가지고 시작해 보겠다. 두 개의 정수, 두 개의 배정도 실수(double)와 두 개의 문자(character) 변수 중에서 큰 값을 찾길 원한다고 하자. 이 경우 아래와 같이 세 개의 오버로딩된 함수를 작성해야 된다.

# 템플릿의 기본

```
int maxValue(int value1, int value2)
{
    if (value1 > value2)
        return value1;
    else
        return value2;
}
```

# 템플릿의 기본

```
double maxValue(double value1, double value2)
{
    if (value1 > value2)
        return value1;
    else
        return value2;
}
```

# 템플릿의 기본

```
char maxValue(char value1, char value2)
{
    if (value1 > value2)
        return value1;
    else
        return value2;
}
```

# 일반화 함수(Generic Function)

```
GenericType max Value(  
    GenericType value1, GenericType value2)  
{  
    if (value1 > value2)  
        return value1;  
    else  
        return value2;  
}
```

# 일반화 함수

C++에서는 일반화 유형을 가지는 함수 템플릿을 정의할 수 있다. 리스트 15.1은 일반화 유형의 두 개의 값 중에서 최대값을 찾는 템플릿 함수를 정의하고 있다.

[GenericMaxValue](#)

Run

# 매개변수 일치

어떤 유형의 두 개 값들 중 큰 값을 반환하는 일반화 max Value 함수는 다음을 규정한다.

- 두 값은 같은 유형을 가진다.
- > 연산자를 사용하여 두 값을 비교할 수 있다.

# 더 좋은 <typename T>

형식 매개변수를 지정하기 위하여 <typename T> 혹은 <class T> 중 어느 하나를 사용할 수 있다. 하지만 <typename T>를 사용하는 것이 더 효과적인데, <typename T>가 형식 매개변수 임을 좀 더 잘 설명해 주고 있고, <class T>는 클래스 선언과 혼동될 수 있기 때문이다.

# 다중 형식 매개변수

템플릿 함수는 한 개 이상의 매개변수를 가질 수 있다. 이 경우, `<typename T1, typename T2, typename T3>`와 같이 괄호(`<>`) 내부에서 콤마로 분리해야 한다.

# 예제: 일반화 정렬(generic sort)

몇 개의 오버로딩된 함수를 작성하지 않고 어떤 유형에 대해서도 동작하는 단 한 개의 템플릿 함수를 정의하면 된다. 리스트 15.2는 요소의 배열을 정렬하기 위한 일반화 함수를 정의하고 있다.

[GenericSort](#)

Run

# 일반화 함수 개발

일반화 함수(generic function)를 정의할 때, 비일반화 함수(non-generic function)로 시작해서 디버그하고 테스트 한 후, 일반화 함수로 변환하는 것이 더 효율적이다.

# 클래스 템플릿

## StackOfIntegers

-elements: **int**[100]

-size: **int**

+StackOfIntegers()

+empty(): bool

+peek(): **int**

+push(value: **int**): **int**

+pop(): **int**

+getSize(): **int**

## Stack<**T**>

-elements: **T**[100]

-size: int

+Stack()

+empty(): bool

+peek(): **T**

+push(value: **T**): **T**

+pop(): **T**

+getSize(): int

[GenericStack](#)

[TestGenericStack](#)

Run

# 컴파일 쟁점

GenericStack.h는 하나의 파일 안에 클래스 선언과 클래스 구현이 모두 작성되어 있다. 일반적으로 클래스 선언과 클래스 구현은 두 개의 분리된 파일로 만든다. 하지만 클래스 템플릿에 대해서는 그 둘을 함께 놓는 것이 더 안전한데, 어떤 컴파일러는 클래스 선언과 클래스 구현을 분리하여 컴파일 할 수 없기 때문이다.

# 기본 유형

C++는 클래스 템플릿에서 형식 매개 변수에 대해 *기본 유형(default type)*을 할당할 수 있다. 예를 들어, 다음과 같은 일반화 Stack 클래스에서 기본 유형으로써 int를 할당할 수 있다:

```
template<typename T = int>  
class Stack  
{  
    ...  
};
```

# 기본 유형

아래와 같이 기본 유형을 사용하여 객체를 선언할 수 있다.

```
Stack<> stack; // 스택은 int 값을 위한 스택임.
```

기본 유형은 클래스 템플릿에서만 사용 가능하며, 함수 템플릿에서는 기본 유형을 사용할 수 없다.

# 비형식 매개변수

템플릿 접두어에서 형식 매개변수와 함께 *비형식 매개변수*(*nontype parameter*)도 사용할 수 있다. 예를 들면, 다음과 같은 Stack 클래스에 대한 매개변수로서 배열 용량(*capacity*)을 선언할 수 있다:

```
template<typename T, int capacity>
class Stack
{
    ...
private:
    T elements[capacity];
    int size;
};
```

# 템플릿과 상속

비템플릿 클래스(nontemplate class)는 클래스 템플릿으로부터 파생될 수 있고, 클래스 템플릿도 비템플릿 클래스로부터 파생될 수 있다. 또한 클래스 템플릿은 클래스 템플릿 으로부터 파생될 수 있다.

# 템플릿 클래스 프렌드

프렌드(friend)는 템플릿과 비템플릿 클래스에서 모두 똑같이 사용된다.

# 정적 멤버

템플릿 클래스에서 *정적 멤버*(*static members*)를 정의할 수 있다. 각 템플릿은 정적 데이터 필드의 자기 자신의 복사본을 가진다.

# Stack 클래스 개선

Stack 클래스에는 문제점이 있다. 스택의 요소는 100개의 고정 크기를 가지는 배열 내에 저장된다 (리스트 15.3의 16번 줄 참조). 따라서 스택에 100개보다 더 큰 수의 요소는 저장할 수 없다. 요소의 수를 100보다 더 큰 수로 변경할 수도 있지만, 이는 실제 스택이 작다면 공간을 낭비하는 결과가 될 것이다. 이 문제를 해결하는 한 방법은 필요할 때만 메모리를 더 많이 할당하는 것이다.

[ImprovedStack](#)