Chapter 18

템플릿

판화가 다양한 물감으로 똑같은 그림을 찍어낼 수 있듯이,
C++는 템플릿을 이용하여 타입만 다른 함수와 클래스를 생성하는 문법을
제공한다. 이번 장에서는 템플릿을 제대로 이해하여 코드를
복사-붙여넣기 하는 수준에서 한 단계 도약하여 템플릿 메타 프로그래밍을
할 수 있는 방법을 살펴본다.

1 템플릿의 개요

2 함수 템플릿

3 클래스 템플릿

4 가변 인자 템플릿

++ 연습 문제

++ 프로그래밍 문제

템플릿의 개요

C 언어에서 C++로 발전하면서 돋보인 특징 중 하나가 바로 함수 중복 정의였다.

[예제 18-1] 함수 중복 정의에 의한 다형성

```
1  // ------------------------------------------------
2  // C 언어
3  int add_int(int a, int b)
4  {
5     return a + b;
6  }
7
8  double add_double(double a, double b)
9  {
10    return a + b;
11 }
12
13 // ------------------------------------------------
14 // C++
15 int add(int a, int b)
16 {
17    return a + b;
18 }
19
20 double add(double a, double b)
21 {
22    return a + b;
23 }
```

[예제 18-1]은 함수 중복 정의가 가져온 변화를 보여준다. 변화의 가장 큰 이점은 함수 사용자들이 대부분 누릴 수 있었다. 기존 C 언어 시절에는 인자 타입에 따른 함수 이름을 구분하는 불편함이 있었으나 C++에서는 오직 add란 이름 하나만 알면 그만이다. 물론 함수 작성자 입장에서도 개별 함수 이름을 정할 때 고민하지 않아서 좋긴 하지만 여전히 비슷한 기능을 하는 함

수를 타입별로 만드는 수고는 줄어들 수 없었다. add 함수의 본체라고 할 수 있는 17행, 22행을 비교하면 완전히 똑같음을 알 수 있다. 사람은 똑같은 작업을 반복하는 것에서 지루함과 무료함을 느끼게 된다. 그리고 보통 새로운 방법을 도입하여 그로부터 벗어나곤 한다. 이런 이유로 템플릿이 도입되었다.

템플릿Template의 사전적 의미는 '형판', '틀', '견본'이다. C++에서는 타입에 상관없이 공통의 알고리즘이나 자료구조에 대해서 코드 조각 틀을 만들어놓고 필요할 때 실제 타입이 적용된 코드를 찍어내는 방식을 사용한다. 미술에 비유하면 판화라고 생각할 수 있다. 형상을 그린 판을 만들어놓고, 다양한 색상의 물감으로 판화를 찍어낼 수 있다. 여기서 판을 템플릿, 색상은 타입으로 대응시킬 수 있다.

C++ 템플릿은 두 가지 경우로 사용된다. 함수 템플릿과 클래스 템플릿이다.

함수 템플릿

2.1 함수 템플릿 정의

```
template<typename T1, ..., typename TN>
RET_TYPE FUNC_NAME(ARG_LIST)
{
   FUNC_BODY
}
```

함수 템플릿은 template〈〉로 시작한다. 〈〉 사이에 템플릿에서 사용할 임의의 타입 이름을 인자로 나열한다. 인자로 전달되는 타입 이름을 템플릿 타입 이름이라고 하며 식별자로서 마음대로 정할 수 있지만 일반적인 관례로 T, _Tx, _Ty, T1 등과 같이 가능하면 짧게 만든다.

템플릿 타입 이름을 나타내기 위하여 typename이라는 키워드를 사용하는데 구형 컴파일러는 typename 대신 class를 사용하였다. typename, class 모두 사용 가능하지만 가능하면 typename을 사용하는 편이 좋다.

template〈〉 다음에는 함수 정의와 똑같은 형식의 함수 틀이 오게 되는데, 함수 틀은 템플릿 타입 이름을 사용하여 함수를 정의한 것이다. 따라서 함수의 반환 타입이나 인자 타입, 함수 본체에서 사용되는 모든 타입은 템플릿 타입 이름으로 대체할 수 있다.

[예제 18-2] add 함수 템플릿

```
1  template<typename T>
2  T add(T a, T b)
3  {
4     return a + b;
5  }
```

[예제 18-2]는 기존 add 함수를 함수 템플릿으로 작성한 것이다. 기존 add 함수의 반환 타입과 인자 타입인 int, double이 템플릿 타입 이름 T로 치환되었음을 알 수 있다. 그 외에는 완전한 함수 정의와 일치한다.

2.2 함수 템플릿의 구체화

이번에는 함수 템플릿을 실제로 함수처럼 사용해보자.

[예제 18-3] 함수 템플릿의 사용

```
1  #include <iostream>
2  using namespace std;
3
4  void main()
5  {
6     cout << add<int>(1, 2) << endl;     // 3 출력
7     cout << add<double>(1.2, 2.4) << endl;     // 3.6 출력
8  }
```

[예제 18-3]의 6행, 7행은 함수 템플릿을 사용하는 방법을 보여준다. add 바로 옆에 〈int〉, 〈double〉이 붙어 있는데 이때 함수 템플릿 타입 이름 T가 int, double로 치환되면서 실제 타입에 대한 add 함수가 정의되고 호출되도록 코드가 변경된다. 이렇게 템플릿으로부터 실제 함수가 정의되는 것을 '구체화'라고 부른다. 실제 템플릿이 구체화된 코드는 [예제 18-4]와 같다.

[예제 18-4] 함수 템플릿의 구체화

```
1  #include <iostream>
2  using namespace std;
3
4  int add(int a, int b)
5  {
6     return a + b;
7  }
8
9  double add(double a, double b)
10 {
11    return a + b;
12 }
13
14 void main()
15 {
16    cout << add(1, 2) << endl;    // 3 출력
17    cout << add(1.2, 2.4) << endl;    // 3.6 출력
18 }
```

```
3
3.6
```

[예제 18-4]는 템플릿이 구체화된 코드를 보여준다. 템플릿 타입 이름 T가 실제 int와 double로 치환되면서 함수 두 개가 정의된 것을 보여준다. 템플릿은 말 그대로 틀일 뿐이고, 실제 사용되기 위해서는 틀에서 실제 함수를 찍어내는 과정이 필요하다. '구체화'란 템플릿으로부터 실제 함수를 찍어내는 과정이라고 할 수 있다. 그렇다면 구체화는 언제 이루어질까? 컴파일 시점에 컴파일러에 의해서 구체화가 이루어진다. 따라서 전처리기의 전처리 과정과 비슷하다고 할 수 있다.

함수 템플릿이 컴파일러에 의해서 구체화되어 실제 생성된 함수를 '템플릿 함수'라고 부른다. 즉, 4행, 10행의 두 개의 add 함수가 바로 템플릿 함수인 것이다. 간단하게 정리하면 함수 템플릿이 구체화된 결과물이 템플릿 함수이다. 마치 판과 판화의 관계라고 할 수 있다. 예제에서는 구체화된 함수 이름을 add로 표시하였지만 설명의 편의상 그대로 사용한 것이고, 실제 컴파일러 구현에 따라서 구체화되는 함수 이름에는 보통 템플릿 타입 이름을 추가하여 함수를 구분

한다. 즉, add〈int〉, add〈double〉과 같은 이름을 갖는다. 이렇게 이름을 다르게 붙이는 이유는 템플릿 함수는 함수 타입은 동일하지만 구현만 달라지는 경우도 존재할 수 있기 때문이다.

템플릿 함수는 실제 사용되는 경우만 정의된다. 만약 전혀 사용되지 않는 함수라면 불필요하게 코드 용량만 차지하고 구체화하는 시간만 낭비되기 때문이다. 즉, 컴파일러는 실제 사용되는 템플릿 함수의 템플릿 타입 이름을 결정하여 구체화한다. 그렇다면 컴파일러는 어떻게 템플릿 타입 이름을 결정하는 것일까? 두 가지 방식이 있는데 사용자가 명시적으로 지정하는 방식과 암시적으로 추론하는 방식이다.

명시적 타입 지정

```
add<int>(1, 2)
add<double>(1.2, 2.4)
```

앞의 예제에서 사용된 것처럼 함수 바로 뒤에 〈〉 안에 구체화할 템플릿 타입 이름을 명시적으로 지정한다. 이때 컴파일러는 템플릿 타입 이름 T를 int와 double로 각각 치환하여 구체화하면서 템플릿 함수를 정의한다.

암시적 타입 추론

명시적으로 타입 지정이 되어 있지 않아도 컴파일러는 전달되는 인자의 타입을 추론하여 템플릿 타입 이름을 결정할 수 있다. 이것은 함수 중복 정의에서 실인자의 타입을 추론하여 적절한 함수를 찾는 것과 같다.

[예제 18-5] 암시적 타입 추론

```
1  #include <iostream>
2  using namespace std;
3
4  template<typename T>
5  T add(T a, T b)
6  {
7     return a + b;
8  }
```

```
9
10  void main()
11  {
12     cout << add(1, 2) << endl;     // OK
13     cout << add(1.2, 2.4) << endl;    // OK
14     cout << add(1, 2.2) << endl;    // Error
15  }
```

[예제 18-5]는 암시적 타입 추론을 통한 템플릿 구체화를 보여준다. 먼저 12행을 살펴보자. add 인자로 1, 2가 전달된다. 1, 2는 컴파일러에 의해서 int로 추론된다. 따라서 템플릿 타입 이름 T를 int로 치환하여 구체화된다. 13행도 마찬가지다. add 인자로 1.2, 2.4가 전달되는데 컴파일러는 double로 추론하여 템플릿 타입 이름 T를 double로 치환하여 구체화한다. 14행은 주의해서 보아야 한다. add 인자로 1과 2.2가 전달된다. 컴파일러는 두 인자로부터 하나의 타입을 추론할 수가 없다. 따라서 템플릿 타입 이름 T를 특정한 타입으로 구체화할 수 없기 때문에 오류가 발생한다.

2.3 템플릿 타입 이름

템플릿 타입 이름은 함수 템플릿에서 컴파일 가능한 타입만 가능하다.

[예제 18-6] 컴파일이 가능한 템플릿 타입 이름

```
1   class CTest
2   {
3   public:
4      int m_Value = 0;
5   };
6
7   template<typename T>
8   T add(T a, T b)
9   {
10     return a + b;
11  }
12
```

```
13  void main()
14  {
15     int i1, i2;
16     i1 = i2 = 1;
17     add(i1, i2);    // OK
18
19     CTest t1, t2;
20     add(t1, t2);    // Error
21  }
```

[예제 18-6]의 17행은 add의 인자 i1, i2가 int 타입이므로 T를 int로 치환하여 템플릿 함수가 정의되고 호출도 문제없이 잘 된다. 반면 20행은 조금 상황이 다르다. add의 인자로 클래스 CTest의 객체 t1, t2가 전달된다. 따라서 T가 CTest로 치환된 템플릿 함수가 정의된다. 문제는 구체화된 템플릿 함수가 컴파일이 가능하지 않다는 데 있다.

[예제 18-7] T = CTest 구체화 템플릿 함수

```
1  CTest add(CTest a, CTest b)
2  {
3     return a + b;   // Error
4  }
```

[예제 18-7]이 바로 템플릿 타입 이름 T가 CTest로 치환된 템플릿 함수이다. 함수에서 문제가 되는 부분은 3행이다. 클래스 CTest는 더하기(+) 연산에 대한 동작이 정의되지 않았다. 따라서 이 문제를 해결하기 위해서는 CTest가 더하기(+) 연산이 가능하도록 연산자 재정의를 해야 한다.

[예제 18-8] T = CTest가 가능한 CTest

```
1  class CTest
2  {
3  public:
4     CTest& operator + (CTest& obj)
5     {
6        m_Value += obj.m_Value;
7        return *this;
```

```
    }

    int m_Value = 0;
};

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

void main()
{
    CTest t1, t2;
    add(t1, t2);    // OK
}
```

[예제 18-8]의 4~8행에 더하기(+) 연산이 가능하도록 연산자 재정의를 추가했다. 실제 템플릿 함수가 정의되어도 더 이상 컴파일 오류가 발생하지 않는다.

템플릿 타입 이름은 기본 타입을 지정할 수 있다. 마치 함수의 기본 인자와 같다. 기본 인자와 마찬가지로 마지막 타입 이름부터 기본 타입을 지정할 수 있다. 기본 타입이 지정된 경우 템플릿 타입 이름을 명시적으로 지정할 경우 생략 가능하다.

[예제 18-9] T = CTest가 가능한 CTest

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

template<typename T1, typename T2 = int>
T2 Cast(T1 a)
{
    return (T2)a;
}

void main()
```

```
11  {
12      cout << Cast<double>(3.3) << endl;     // 3 출력
13      cout << Cast<double, double>(3.3) << endl;     // 3.3 출력
14  }
```

[예제 18-9]의 함수 템플릿은 인자 타입 T1을 T2로 타입 변환하여 반환한다. 4행의 T2는 기본 타입이 int로 지정되어 있다. 12행에서 명시적으로 지정된 타입은 〈double〉이다. 따라서 T1은 double로 치환된다. T2의 경우 명시적으로 지정된 타입이 없으므로 기본 타입으로 지정된 int로 치환된다. 따라서 인자 3.3에 대해서 int로 타입 변환하므로 3이 출력된다. 13행은 명시적으로 지정된 타입이 〈double, double〉이다. 따라서 T1은 double로 치환되고, T2도 double로 치환된다. 그러므로 3.3은 그대로 3.3으로 출력된다.

2.4 특수화

구체화된 템플릿 함수들은 각각의 타입만 다를 뿐 함수 자체의 동작은 동일하다. 그러나 특정 타입에 대해서만 동작을 다르게 정의할 필요도 있다. 이런 경우 사용하는 템플릿 용법을 '특수화'라고 한다.

[예제 18-10] Sigma 함수

```
1   #include <iostream>
2   using namespace std;
3
4   template<typename T>
5   T Sigma(T a)
6   {
7       if(a == 1)
8       {
9           return 1;
10      }
11
12      return a + Sigma(a - 1);
13  }
14
```

```
15  void main()
16  {
17     cout << Sigma(10) << endl;    // 5 출력
18     cout << Sigma(10.1) << endl;  // Runtime Error
19  }
```

[예제 18-10]의 함수 템플릿은 시그마 함수를 나타낸다. 1부터 시작하여 인자로 입력된 값까지 총합을 구해서 반환한다. 시그마 함수의 특성상 인자의 타입은 정수만 가능하다. 따라서 타입으로 따지면 char, short, int, __int64 등이 가능하다. 정수 타입이 아닌 float, double은 인자의 타입으로 부적절하다. 실제로 Sigma 함수 템플릿의 구현부를 살펴볼 경우 인자의 값이 정수가 아닐 경우 9행에 진입되지 않으므로 무한 재귀 호출이 발생한다. 따라서 18행에서 인자로 10.1이 입력될 경우 런타임 오류가 발생한다. 이런 경우를 방지하기 위하여 특정 타입에 한해서 특수화를 사용할 수 있다.

[예제 18-11] 함수 템플릿 특수화

```
1   #include <iostream>
2   using namespace std;
3
4   template<typename T>
5   T Sigma(T a)
6   {
7      if(a == 1)
8      {
9         return 1;
10     }
11
12     return a + Sigma(a - 1);
13  }
14
15  template<>
16  double Sigma<double>(double a)
17  {
18     cout << "지원하지 않는 타입입니다." << endl;
19     return 0;
```

```
20  }
21
22  void main()
23  {
24     cout << Sigma(10) << endl;    // 5 출력
25     cout << Sigma(10.1) << endl;    // 0 출력
26  }
```

```
55
지원하지 않는 타입입니다.
0
```

[예제 18-11]은 Sigma 함수 템플릿에 대해서 double 타입 특수화를 한 것이다. 13~20행이 특수화된 함수 템플릿이다. 특수화는 먼저 15행처럼 template〈〉로 시작하며 16행의 함수 선언부에는 템플릿 타입 이름 T를 모두 특정 타입으로 변경하면 된다. 또한 특수화된 특정 타입을 명시적으로 함수 이름 바로 뒤에는 〈〉 안에 기입할 수도 있다. 물론 특정 타입이 추론 가능하면 생략해도 무방하다. 그 이후 함수 본체 부분은 자유롭게 작성할 수 있다.

25행에서 인자 10.1은 double 타입이므로 컴파일러는 double에 대해서 특수화된 함수 템플릿을 구체화하여 템플릿 함수를 정의한다. 컴파일러는 함수 템플릿 중에서 특수화 템플릿을 먼저 선택하며 특수화 템플릿이 없을 경우 일반 템플릿이 선택된다.

2.5 멤버 함수 템플릿

멤버 함수도 함수이므로 템플릿을 적용할 수 있다.

[예제 18-12] 멤버 함수 템플릿

```
1  #include <iostream>
2  using namespace std;
3
4  class CTest
5  {
6  public:
7     template<typename T>
8     T Cast()
```

```
 9     {
10        return (T)m_Value;
11     }
12
13     double m_Value;
14  };
15
16  void main()
17  {
18     CTest t;
19     t.m_Value = 3.14;
20     cout << t.Cast<int>() << endl;
21     cout << t.Cast<double>() << endl;
22  }
```

```
3
3.14
```

[예제 18-12]의 7~11행이 바로 멤버 함수 템플릿을 보여준다. 20행, 21행처럼 멤버 함수가 호출되는 경우 타입에 따라서 클래스의 멤버 함수가 구체화되어 정의되는 구조이다. 함수 템플릿과 특별히 다를 것이 없으며 단지 멤버 함수라는 것만 차이가 있다.

3 클래스 템플릿

3.1 클래스 템플릿 정의

함수 템플릿이 함수를 찍어내듯이 클래스 템플릿은 클래스를 찍어낸다. 최종 구체화된 산출물이 함수에서 클래스로 바뀐 것뿐이다.

```
template<typename T1, ..., typename TN>
class ClassName [ : BASE_LIST ]
{
   CLASS BODY
};
```

클래스 템플릿도 template〈〉로 시작한다. 〈〉 사이에 템플릿에서 사용할 임의의 타입 이름을 인자로 나열한다. template〈〉 다음에는 클래스 정의와 똑같은 형식의 클래스 틀이 오게 되며 클래스 정의에서 사용되는 모든 타입에 대해서 템플릿 타입 이름으로 대체할 수 있다.

[예제 18-13] 클래스 템플릿

```
1  template<typename T>
2  class CTest
3  {
4  public:
5     T GetValue()
6     {
7        return m_Value;
8     }
9
10    T m_Value;
11 };
```

[예제 18-13]은 클래스 템플릿 정의를 보여준다. 함수 템플릿과 크게 다를 것이 없고, 템플릿 타입 이름 T를 클래스 정의의 어떤 타입에도 사용할 수 있다. 5행에서는 멤버 함수 GetValue의 반환 타입으로 T를 사용하고, 10행에서는 멤버 데이터 m_Value의 타입으로 T를 사용한다.

3.2 클래스 템플릿의 구체화

[예제 18-14] 클래스 템플릿의 사용

```
1  #include <iostream>
2  using namespace std;
3
4  void main()
5  {
6     CTest<int> ti;
7     ti.m_Value = 3.14;
8     cout << ti.GetValue() << endl;   // 3 출력
9
```

```
10     CTest<double> td;
11     td.m_Value = 3.14;
12     cout << td.GetValue() << endl;   // 3.14 출력
13 }
```

[예제 18-14]의 6행, 10행은 클래스 템플릿을 사용하는 방법을 보여준다. CTest 바로 옆에 〈int〉, 〈double〉이 붙어 있는데 이때 클래스 템플릿 타입 이름 T가 int, double로 치환되면서 실제 타입에 대한 클래스가 정의되도록 코드가 변경된다. 이렇게 템플릿으로부터 실제 클래스가 정의되는 것을 구체화라고 부른다. 실제 템플릿이 구체화된 코드는 [예제 18-15]와 같다.

[예제 18-15] 클래스 템플릿의 구체화

```
1  #include <iostream>
2  using namespace std;
3
4  class CTest_A
5  {
6  public:
7     int GetValue()
8     {
9        return m_Value;
10    }
11
12    int m_Value;
13 };
14
15 class CTest_B
16 {
17 public:
18    double GetValue()
19    {
20       return m_Value;
21    }
22
23    double m_Value;
```

```
};

void main()
{
   CTest_A ti;
   ti.m_Value = 3.14;
   cout << ti.GetValue() << endl;

   CTest_B td;
   td.m_Value = 3.14;
   cout << td.GetValue() << endl;
}
```

```
3
3.14
```

[예제 18-15]의 구체화는 조금 이상하다. 클래스 이름이 CTest에서 CTest_A, CTest_B로 바뀌었다. 바뀐 이름은 필자가 정한 것이고 컴파일러 내부의 처리를 설명하기 위한 것이다. 클래스 정의가 다르다면 클래스 이름도 달라야 한다. 컴파일러는 내부적으로 클래스 템플릿을 구체화할 경우 구체화된 클래스를 구분하여 처리한다. 템플릿 함수처럼 클래스 템플릿이 구체화된 클래스를 '템플릿 클래스'라고 한다.

함수 템플릿의 경우 전달되는 인자의 타입을 추론하여 암시적으로 구체화가 가능하였다. 그러나 클래스 템플릿의 경우는 전달되는 인자가 존재하지 않으므로 명시적 타입 지정 방식만 허용된다.

3.3 멤버 함수의 외부 정의

클래스 템플릿을 정의할 때 생성자, 소멸자, 멤버 함수를 클래스 정의 외부에 할 수 있다. 이 경우 template 〈〉을 사용하여 소속 클래스를 명확하게 밝혀야 한다.

[예제 18-16] 멤버 함수의 외부 정의

```
template<typename T>
class CTest
{
```

```
4  public:
5     CTest();
6     ~CTest();
7     void Func();
8     T Get();
9
10    T m_Value;
11 };
12
13 template<typename T>
14 CTest<T>::CTest()
15 {
16 }
17
18 template<typename T>
19 CTest<T>::~CTest()
20 {
21 }
22
23 template<typename T>
24 void CTest<T>::Func()
25 {
26 }
27
28 template<typename T>
29 T CTest<T>::Get()
30 {
31    return m_Value;
32 }
```

[예제 18-16]은 생성자, 소멸자, 멤버 함수를 클래스 템플릿 외부에 정의하는 것을 보여준다. 보통 일반 클래스의 멤버 함수를 외부 정의할 경우 해당 멤버 함수가 어떤 클래스 소속인지를 밝히기 위하여 범위(::) 연산자를 사용하여 클래스를 밝혔다. 마찬가지로 클래스 템플릿이 구체화되어 각각 템플릿 클래스가 여러 개가 정의될 경우 멤버 함수도 각각의 클래스에 포함되도록

정의되어야 한다. 그러므로 각 멤버 함수의 외부 정의마다 template 〈〉가 반드시 필요한 것이다. 7행의 멤버 함수 Func를 보면 템플릿 타입 이름 T와는 전혀 연관이 없기 때문에 template 〈〉를 사용하지 않아도 될 것 같지만 Func 함수 안에서 T 타입을 사용할지의 여부는 알 수 없으므로 무조건 template 〈〉를 사용해야만 한다.

클래스 템플릿 멤버 함수의 외부 정의를 할 경우 주의할 점이 있다. 클래스 템플릿이 작성되는 파일과 멤버 함수가 외부 정의되는 파일을 가능하면 일치시켜야 한다는 점이다. 먼저 일반 클래스와 비교하면서 왜 그런지 이유를 확인해보자. 일반적으로 일반 클래스의 경우 클래스 정의는 헤더 파일에 있고, 멤버 함수의 정의는 소스 파일에 존재한다. 그리고 실제 클래스 객체를 사용하는 곳은 다른 소스 파일인 경우가 대부분이다. 즉, [예제 18-17]과 같은 구조이다.

[예제 18-17] 클래스의 정의, 구현, 사용

```
1  // -----------------
2  // <A.h>
3  class CTest
4  {
5  public:
6     void Func();
7  };
8
9  // -----------------
10 // <A.cpp>
11 #include "A.h"
12 void CTest::Func()
13 {
14 }
15
16 // -----------------
17 // <Main.cpp>
18 #include "A.h"
19 void main()
20 {
21    CTest t;
```

```
22     t.Func();
23 }
```

[예제 18-17]은 일반적인 클래스의 정의와 멤버 함수의 외부 정의 그리고 클래스 객체의 생성 및 사용이 각각의 헤더, 소스 파일에 존재함을 보여준다. 클래스 CTest는 헤더 파일 A.h에 정의되어 있고, 멤버 함수 Func는 소스 파일 A.cpp에 정의되어 있다. 실제 CTest 객체를 정의하고 멤버 함수를 호출하는 코드는 소스 파일 Main.cpp에 있다. 이렇게 클래스 관련 코드가 각 파일에 나뉘어도 문제가 없는 이유는 빌드 과정이 컴파일과 링크로 이루어지기 때문이다. 22행의 객체 t를 기준으로 Func 함수를 호출하는 코드는 링크 과정에서 실제 Func 함수의 구현부와 연결된다. 그렇다면 이번에는 CTest를 클래스 템플릿으로 변경해보자.

[예제 18-18] 템플릿 클래스의 정의, 구현, 사용

```
1  // -----------------
2  // <A.h>
3  template<typename T>
4  class CTest
5  {
6  public:
7     void Func();
8  };
9
10 // -----------------
11 // <A.cpp>
12 #include "A.h"
13 template<typename T>
14 void CTest<T>::Func()
15 {
16 }
17
18 // -----------------
19 // <Main.cpp>
20 #include "A.h"
21 void main()
```

```
22  {
23     CTest<int> t;
24     t.Func();     // Link Error
25  }
```

[예제 18-18]은 이전 소스의 클래스를 클래스 템플릿으로 변경한 것이지만 24행에서 링크 오류가 발생한다. 왜 오류가 발생한 것일까? 바로 A.cpp 파일의 멤버 함수가 제대로 정의되지 않았기 때문이다. 템플릿 구체화는 컴파일 과정에서 진행되는데 실제 사용되는 타입에 대해서만 구체화가 이루어진다. 따라서 A.cpp를 컴파일할 시점에는 23행처럼 〈int〉가 지정되었는지를 알 수 없기 때문에 멤버 함수가 정의될 수 없다. 따라서 24행에서 링크를 할 경우 Func 함수를 찾을 수 없는 것이다.

이런 이유로 클래스 템플릿을 작성하는 경우 클래스 템플릿 정의와 멤버 함수의 정의도 보통 같은 헤더 파일에 넣게 된다. 이렇게 할 경우 소스 파일에서 컴파일이 진행될 경우 클래스의 멤버 함수 정의까지 구체화될 수 있기 때문이다.

3.4 특수화

함수 템플릿 특수화처럼 클래스 템플릿 또한 특수화를 할 수 있다. 한마디로 특정 타입에 대해서 클래스를 새롭게 정의하는 것과 같다.

[예제 18-19] 클래스 템플릿 특수화

```
1  #include <iostream>
2  using namespace std;
3
4  template<typename T>
5  class CTest
6  {
7  public:
8     void Func()
9     {
10       cout << sizeof(T) << endl;
```

```
11     }
12  };
13
14  template<>
15  class CTest<double>
16  {
17  public:
18     double PI()
19     {
20        return 3.14;
21     }
22  };
23
24  void main()
25  {
26     CTest<char> t1;
27     t1.Func();      // 1 출력
28
29     CTest<int> t2;
30     t2.Func();      // 4 출력
31
32     CTest<double> t3;
33     cout << t3.PI() << endl;    // 3.14 출력
34     t3.Func();  // Error
35  }
```

[예제 18-19]의 클래스 템플릿 CTest〈T〉의 멤버 함수 Func는 타입 T의 크기를 sizeof 연산자를 이용하여 출력한다. 그러나 double 타입에 대하여 특수화된 클래스 템플릿 CTest〈double〉에는 멤버 함수 Func가 존재하지도 않으며 전혀 다른 멤버 함수 PI가 정의되어 있다. 즉, 특수화된 클래스는 완전히 별개의 클래스라고 할 수 있다. 따라서 33행의 t3의 타입은 CTest〈double〉이므로 정의되지 않은 멤버 함수 Func를 호출할 수 없다.

컴파일러는 특수화된 클래스 템플릿에 우선권을 준다.

가변 인자 템플릿

가변 인자 함수처럼 템플릿에도 타입 이름의 개수를 가변적으로 전달할 수 있다. 가변 인자 템플릿을 사용하기 위해서는 몇 가지 기호 및 개념을 이해해야 한다.

4.1 가변 인자 템플릿 기호

[예제 18-20] 가변 인자 템플릿

```
1  #include <iostream>
2  using namespace std;
3
4  int Add(int a1 = 0, int a2 = 0, int a3 = 0, int a4 = 0)
5  {
6     return a1 + a2 + a3 + a4;
7  }
8
9  template<typename... ARGS>
10 int Sum(ARGS... args)
11 {
12    return Add(args...);
13 }
14
15 void main()
16 {
17    cout << Sum(1, 2, 3) << endl;
18 }
```

[예제 18-20]의 4행 Add 함수는 가변 인자 템플릿을 테스트하기 위한 용도로 만든 함수이다. 최대 인자 4개까지 넣을 수 있으며 인자들의 합을 반환한다. 이제 본격적으로 가변 인자 템플릿을 보면, 9~13행이 바로 가변 인자 템플릿을 사용한 함수 템플릿이다. 가변 인자 템플릿에서 제일 중요한 부분이 바로 9행의 〈typename... ARGS〉이다. 여기서 typename...이 바로 가변 인자 템플릿을 나타낸다. 바로 뒤의 ARGS는 가변 인자들의 타입 이름을 나타내는 식별자이

며, ARGS는 이름을 바꿀 수 있으나 관례상 ARGS가 많이 사용된다. 이제 10행 함수 Sum의 매개 변수를 살펴보자. 역시 낯선 표현이 등장하는데 (ARGS... args)에서 args가 실제 전달되는 가변 인자를 나타낸다. args를 보통 parameter pack이라고 하는데, 가변 인자 템플릿을 사용하기 위해서 가장 핵심적인 요소라고 할 수 있다.

12행을 보면 이미 정의한 Add 함수의 인자로 args를 넘기는데 역시 표현이 낯설다. args...는 가변 인자들을 그대로 전달한다는 의미이다. 실제로 가변 인자 템플릿이 어떻게 변환되는지 살펴보자. 17행의 Sum(1, 2, 3)과 같은 호출 구문이 있을 경우 컴파일러는 인자 (1, 2, 3)을 통해서 인자의 개수가 3개인 것을 알 수 있다. 이때 컴파일러는 가변 인자 템플릿을 일반 템플릿으로 변환하는 과정을 거친다. 이때 각 기호들이 치환된 결과는 다음과 같다.

```
9행: <typename... ARGS> → <typename T1, typename T2, typenameT3>
10행: (ARGS... args) → (T1 arg1, T2 arg2, T3, arg3)
12행: (args...) → (arg1, arg2, arg3)
```

위의 변환에 따른 일반 템플릿은 [예제 18-21]과 같다.

[예제 18-21] 일반 템플릿으로 변환

```
1  template<typename T1, typename T2, typename T3>
2  int Sum(T1 arg1, T2 arg2, T3 arg3)
3  {
4     return Add(arg1, arg2, arg3);
5  }
```

[예제 18-21]이 바로 가변 인자 템플릿이 일반 템플릿으로 변환된 모습이다. 지금까지 배운 내용으로 보면 어려움이 없을 것이다. (1, 2, 3)이 전달되면 암시적 타입 추론에 의해서 T1, T2, T3이 모두 int로 치환되면서 구체화될 것이고 템플릿 함수에 (1, 2, 3)이 전달되면 6이 출력된다. 결국 가변 인자 템플릿에서 가장 중요한 것은 가변 인자 템플릿에서 사용되는 기호들이 어떻게 일반 템플릿을 구성하기 위하여 변환되는지를 이해하는 것이다.

이번에는 가변 인자 템플릿에서 인자의 개수를 알아내는 방법을 알아보자.

[예제 18-22] sizeof...

```
1  #include <iostream>
2  using namespace std;
3
4  template<typename... ARGS>
5  void Func(ARGS... args)
6  {
7     cout << sizeof...(ARGS) << endl;
8     cout << sizeof...(args) << endl;
9  }
10
11 void main()
12 {
13    Func(1, 2, 3, 4, 5, 6);
14 }
```

[예제 18-22]의 7행에서도 역시 낯선 표현이 등장한다. 바로 sizeof...이다. sizeof 연산자는 피연산자의 타입 크기를 반환하지만 sizeof...는 피연산자로 가변 인자 타입을 대표하는 ARGS나 가변 인자를 대표하는 args를 받아서 가변 인자의 개수를 반환한다. 즉, sizeof와 sizeof...는 전혀 관련이 없다고 생각하면 된다.

4.2 가변 인자의 추출

args...를 이용하여 가변 인자 전체를 다른 함수에 전달할 수는 있으나 가변 인자들을 개별적으로 하나씩 구하기 위해서는 재귀적인 방식을 사용해야 한다.

[예제 18-23] 가변 인자 추출

```
1  #include <iostream>
2  using namespace std;
3
4  void Func()
5  {
6  }
```

```
7
8  template<typename T, typename... ARGS>
9  void Func(T arg, ARGS... args)
10 {
11    cout << arg << endl;
12    Func(args...);
13 }
14
15 void main()
16 {
17    Func(1, 2);
18 }
```

```
1
2
```

[예제 18-23]의 8~13행의 함수 템플릿은 가변 인자를 받아서 한 줄에 하나씩 출력한다. 특이한 점은 8행에서 가변 인자 앞에 타입 이름 T가 추가되었다는 것이다. 마찬가지로 9행에서도 함수 첫 번째 인자로 T arg가 추가되었다. 이것이 바로 가변 인자를 하나씩 추출하기 위한 핵심 트릭이다. 실제로 17행이 어떻게 해석되는지 자세히 살펴보자.

먼저 컴파일러는 17행의 인자 (1, 2)를 통해서 인자의 수가 2개임을 알 수 있다. 그리고 가변 인자 템플릿을 일반 템플릿으로 변환한다.

```
8행: <typename T, typename... ARGS>  → <typename T, typename T1>
9행: Func(T arg, ARGS... args) → Func(T arg, T1 arg1)
12행: Func(args...) → Func(arg1)
```

12행의 Func(arg1)을 통해서 컴파일러는 인자의 수가 1개임을 알 수 있고, 다시 가변 인자 템플릿을 일반 템플릿으로 변환한다.

```
8행: <typename T, typename... ARGS>  → <typename T>
9행: Func(T arg, ARGS... args) → Func(T arg)
12행: Func(args...) → Func()
```

결국 최종 변환된 템플릿 코드는 [예제 18-24]와 같다.

[예제 18-24] 가변 인자 추출 템플릿

```
1  #include <iostream>
2  using namespace std;
3
4  void Func()
5  {
6  }
7
8  template<typename T>
9  void Func(T arg)
10 {
11    cout << arg << endl;
12    Func();
13 }
14
15 template<typename T, typename T1>
16 void Func(T arg, T1 arg1)
17 {
18    cout << arg << endl;
19    Func(arg1);
20 }
21
22 void main()
23 {
24    Func(1, 2);
25 }
```

[예제 18-24]는 일반 함수 템플릿이므로 쉽게 해석할 수 있다. 24행의 인자 (1, 2)를 통해서 암시적 타입 추론을 하여 15~20행 함수 템플릿이 구체화되어 호출된다. 이때 arg는 1이고, arg1은 2가 되어서 arg가 먼저 출력된 뒤에 Func(2)가 호출된다. 역시 인자 2를 통해서 암시적 타입 추론을 하여 8~13행 함수 템플릿이 구체화되어 호출된다. 11행에서 arg를 출력한 뒤에 12행에 인자가 없는 함수 Func가 호출된다. 인자가 없는 Func가 호출되는 것은 상당히 중요한데 바로 재귀적인 호출을 멈추는 역할을 한다. 따라서 아무 인자도 없는 함수 Func가 4~6행에 정의되는 것은 매우 중요하다.

4.3 가변 인자 템플릿 활용

지금까지 배운 가변 인자 템플릿을 이용하여 합계 함수를 만들어보자. 인자의 모든 타입은 int라고 가정하고 인자의 개수에 상관없이 모든 인자의 합을 구해서 반환하는 함수 Sum은 [예제 18-25]와 같다.

[예제 18-25] 가변 인자 합계 함수 Sum

```
1  #include <iostream>
2  using namespace std;
3
4  int Sum()
5  {
6     return 0;
7  }
8
9  template<typename T, typename... ARGS>
10 int Sum(T arg, ARGS... args)
11 {
12    return arg + Sum(args...);
13 }
14
15 void main()
16 {
17    cout << Sum(1, 2, 3, 4, 5) << endl;
18 }
```

[예제 18-25]의 4~6행의 Sum은 재귀 호출을 중지하는 함수이다. 12행에서 첫 인자를 추출하고 나머지 인자들을 Sum에 전달하여 재귀 호출한다. 이 과정은 인자가 없는 Sum 함수가 호출되면서 끝난다.

가변 인자 템플릿은 사실 무척 어렵다. 주로 재귀적인 방식을 사용하기 때문에 코드 자체는 분량이 적어도 의미를 파악하고 코드를 작성하는 시간이 훨씬 많이 든다. 그러나 코드도 간결하게 하면서 시간도 단축할 수 있는 방법이 하나 있는데 그것은 바로 많은 연습이다. 아무리 어려워도 자주 해보면 익숙해질 수밖에 없고, 익숙해지면 어느새 쉬워지는 것이 이치이다.

연습 문제

01 – C++ 템플릿은 두 가지로 사용된다. _____ 템플릿과 _____ 템플릿이다.

02 – 함수 템플릿으로부터 실제 함수가 정의되는 것을 _____라고 한다.

03 – 구체화는 _____에 의해서 _____ 시점에 이루어진다.

04 – 함수 템플릿이 구체화되어 실제 정의된 함수를 _____라고 한다.

05 – 다음 프로그램은 컴파일 오류가 발생한다. 오류의 원인은 무엇이고 어떻게 수정해야 하는가?

```
class CTest
{
public:
   int m_Value = 0;
};

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

void main()
{
   CTest t1, t2;
   add(t1, t2);
}
```

06 – 특정 타입에 대해서만 다른 구체화를 하도록 만드는 템플릿 용법을 _____라고 한다.

07 – 클래스 템플릿으로부터 실제 클래스가 정의되는 것을 _____라고 한다.

08 – 클래스 템플릿이 구체화되어 실제 정의된 클래스를 _____라고 한다.

09 – 클래스 템플릿은 전달되는 인자가 존재하지 않으므로 오직 _____ 방식만 허용된다.

10 – 다음과 같은 프로그램에서 〈typename... ARGS〉, (ARGS... args), (args...)가 각각 어떻게 치환되는지 기술하시오.

```
int Add(int a1 = 0, int a2 = 0, int a3 = 0)
{
   return a1 + a2;
}

template<typename... ARGS>
int Sum(ARGS... args)
{
   return Add(args...);
}

void main()
{
   cout << Sum(1, 2) << endl;
}
```

11 – 가변 인자 템플릿에서 가변 인자의 개수를 알아내는 연산자는 _____이다.

프로그래밍 문제

{1}- 다음 프로그램을 템플릿을 이용하여 간소화하시오.

```
int Square(int arg)
{
   return arg * arg;
}

double Square(double arg)
{
   return arg * arg;
}

void main()
{
   cout << Square(3) << endl;
   cout << Square(3.3) << endl;
}
```

{2}- 다음 프로그램의 출력 결과가 나오도록 템플릿 함수 add를 정의하시오(특수화를 사용).

```
void main()
{
   cout << add(1, 2) << endl;
   cout << add(1.2, 2.4) << endl;
   cout << add("우리", "나라") << endl;
}
```

```
3
3.6
우리나라
```

{3}- 다음 프로그램의 출력 결과가 나오도록 템플릿 클래스 Cast를 정의하시오(특수화 및 operator ( ), atof).

```
void main()
{
   Cast<int> ci;
   cout << ci(3.1) << endl;
   cout << ci(4.2) << endl;
   cout << ci("5.3") << endl;
   cout << ci("6.4") << endl;

   Cast<double> cd;
   cout << cd(3.1) << endl;
   cout << cd(4.2) << endl;
   cout << cd("5.3") << endl;
   cout << cd("6.4") << endl;
}
```

```
3
4
5
6
3.1
4.2
5.3
6.4
```

{4}- 다음 프로그램의 출력 결과가 나오도록 템플릿 함수 makestr을 정의하시오(가변 인자 템플릿, sizeof..., memset, strcat).

```
void main()
{
   cout << makestr('C', '+', '+') << endl;
}
```

```
C++
```