Chapter 10 - 템플릿

생성일	@2025년 11월 23일 오전 12:02
태그	

01) 함수 템플릿

템플릿으로 범용 함수 만들기

- **범용 함수 만들기**
  - 다양한 데이터 형식의 **매개변수를 2개 입력**받아서 덧셈하는 범용 함수
  - 데이터 형식 별로 거의 동일한 코드를 반복해서 만들어야 함
  - 처리할 수 있는 데이터 형식을 추가하려면 **함수도 함께 추가**해야 함
- **템플릿**
  - 함수(또는 클래스) 하나로 다양한 형식의 데이터를 같은 알고리즘으로 처리

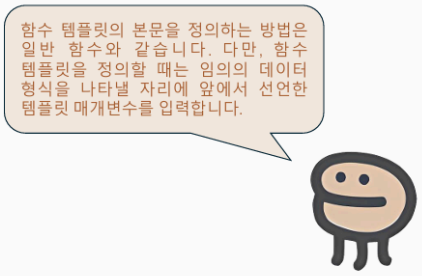
Do it! 실습 함수 템플릿

• ch10/function_template/function_template.cpp

```
#include <iostream>
#include <string>

using namespace std;

// 함수 템플릿 선언과 정의
template <typename T>                          // ← 템플릿 선언
T data_sum(T operand1, T operand2) {           // ← 함수 템플릿 정의
  return operand1 + operand2;
}

int main() {
int data1 = 3, data2 = 5;
double data3 = 4.5, data4 = 8.9;
string data5 = "Hello, ", data6 = "World!";

cout << "정수형 데이터 합: " << data_sum(data1, data2) << endl;
cout << "실수형 데이터 합: " << data_sum(data3, data4) << endl;
cout << "문자열 데이터 합: " << data_sum(data5, data6) << endl;
return 0;
}
```

실행 결과

```
정수형 데이터 합: 8
실수형 데이터 합: 13.4
문자열 데이터 합: Hello, World!
```



- **범용성 확보**
  - **템플릿**은 다양한 데이터 형식을 처리
  - 모든 데이터 형식에 대응할 수 있는 **알고리즘 정의** 필요
- **데이터 형식을 고려하지 않으면 문제가 발생**

템플릿의 인스턴스화

- 함수와 지역 변수는 컴파일러시 스택 메모리 크기가 결정 → 메모리 크기는 컴파일 시점에 결정
  - 템플릿 매개변수에 맞는 **데이터 형식에 맞는 메모리 함수의 크기**가 정해져야 함

1) 템플릿 매개변수의 데이터 형식 추론, 2) 추론된 형식을 사용하는 오브젝트 코드 생성

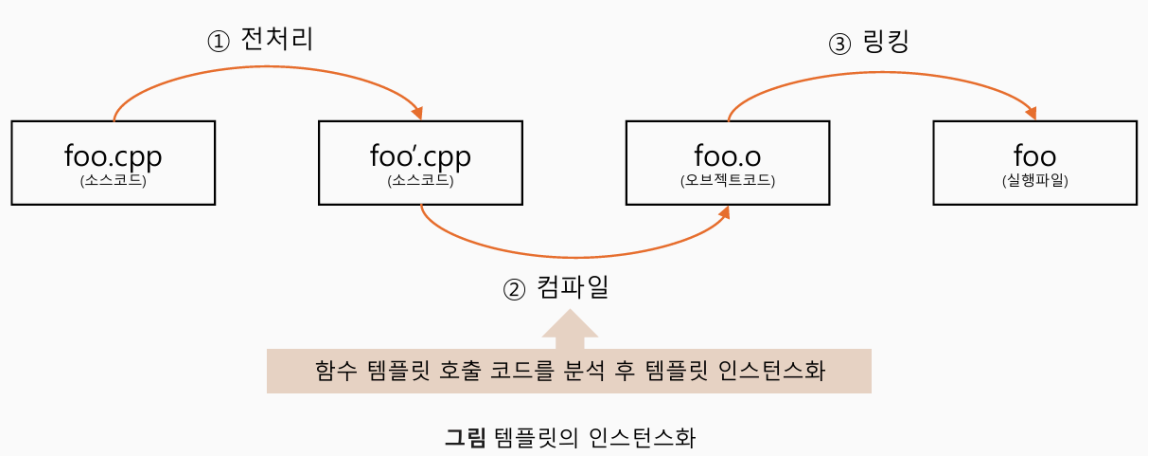


그림 템플릿의 인스턴스화

데이터 형식 추론과 명시적 호출

- 템플릿은 **데이터 형식 추론이 중요한 단계**임
  - 템플릿의 인스턴스화가 진행되기 위해서는 데이터 형식 추론이 중요함
  - 데이터 형식 추론을 할 수 없다면 인스턴스화가 불가능해 오류가 발생함
    - 앞뒤 코드로 데이터 형식 추론이 어렵거나, 암시적 형변환이 어려운 경우 오류가 발생함

```
T data_sum(T operand1, T operade2) {
    return operand1 + operand2
}
... (생략) ...
char data7[] = "Hello, ", data8[] = "New World!";
```

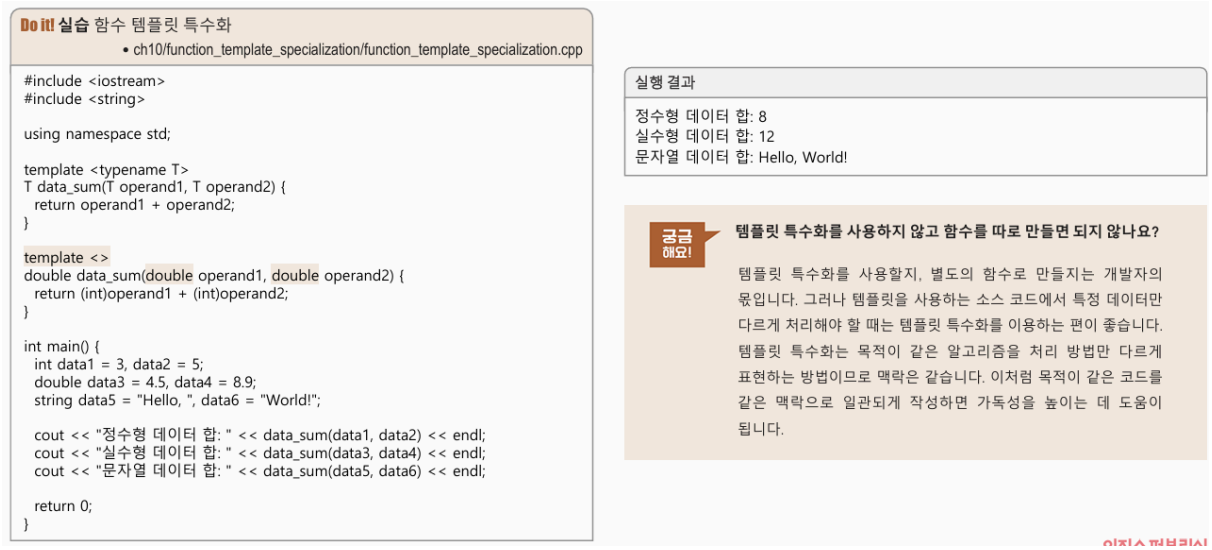
```
cout << "문자 데이터 배열의 합" << data_sum(data7, data
8) << endl;
```

- **명시적 호출**
  - 데이터 형식 추론 과정이 필요 없도록 명시적으로 데이터 형식을 알려 줌
  - 함수 템플릿 호출문에서 인자를 입력하기 전에 꺾쇠 괄호 < >로 데이터 형식을 명시

```
char data7[] = "Hello, ", data8[] = "New World!";
cout << "문자 데이터 배열의 합" << data_sum<string>(data7,
data8) << endl;
```

템플릿 특수화

- 특정 데이터 형식에 대한 예외적인 처리 방법 템플릿 특수화
- 모든 템플릿 매개변수를 특정 데이터 형식으로 지정하는 **'명시적 특수화'**
  - 함수 템플릿은 명시적 특수화만 사용할 수 있음
- 일부 템플릿 매개변수만 특정 데이터 형식으로 지정하는 **'부분 특수화'**
  - 부분 특수화는 함수 템플릿, 클래스 템플릿 모두에 사용할 수 있음



Do it! 실습 함수 템플릿 특수화

• ch10/function_template_specialization/function_template_specialization.cpp

```
#include <iostream>
#include <string>

using namespace std;

template <typename T>
T data_sum(T operand1, T operand2) {
  return operand1 + operand2;
}

template <>
double data_sum(double operand1, double operand2) {
  return (int)operand1 + (int)operand2;
}

int main() {
  int data1 = 3, data2 = 5;
  double data3 = 4.5, data4 = 8.9;
  string data5 = "Hello, ", data6 = "World!";

  cout << "정수형 데이터 합: " << data_sum(data1, data2) << endl;
  cout << "실수형 데이터 합: " << data_sum(data3, data4) << endl;
  cout << "문자열 데이터 합: " << data_sum(data5, data6) << endl;

  return 0;
}
```

실행 결과

```
정수형 데이터 합: 8
실수형 데이터 합: 12
문자열 데이터 합: Hello, World!
```

궁금해요!

템플릿 특수화를 사용하지 않고 함수를 따로 만들면 되지 않나요?

템플릿 특수화를 사용할지, 별도의 함수로 만들지는 개발자의 몫입니다. 그러나 템플릿을 사용하는 소스 코드에서 특정 데이터만 다르게 처리해야 할 때는 템플릿 특수화를 이용하는 편이 좋습니다. 템플릿 특수화는 목적이 같은 알고리즘을 처리 방법만 다르게 표현하는 방법이므로 맥락은 같습니다. 이처럼 목적이 같은 코드를 같은 맥락으로 일관되게 작성하면 가독성을 높이는 데 도움이 됩니다.

02) 클래스 템플릿

템플릿으로 범용 클래스 만들기

- 함수 템플릿에서 사용된 템플릿을 클래스로 확장한 것
  - 멤버 변수 데이터 형식, 멤버 함수의 매개변수 데이터 형식을 템플릿으로 변경
  - 템플릿 선언은 함수 템플릿과 동일

```
template <typename Type1, typename Type2>
class data_package {
public:
    data_package(Type1 first, Type2 second) first(firs
t), second(second) {}
private:
    Type1 first;
    Type2 second;
};
```

  - 객체 생성 시, **템플릿 매개변수에 사용할 데이터 형식을 지정**해 주어야 함

```
int main() {
    data_package<int, double> template_inst1(5, 10.5);
    data_package<string, int> template_inst2("문자열", 1
0);
...(생략)...
```

Do it! 실습 클래스 템플릿 사용

• ch10/class_template/class_template.cpp

```
#include <iostream>
#include <string>

using namespace std;

template <typename Type1, typename Type2>
class data_package {
public:
  data_package(Type1 first, Type2 second) : first(first), second(second){}
  void print_out_element() {
    cout << "첫 번째 요소: " << first << ", 두 번째 요소: " << second
         << endl;
  }
private:
  Type1 first;
  Type2 second;
};

int main() {
  data_package<int, double> template_inst1(5, 10.5);
  data_package<string, int> template_inst2("문자열", 10);

  template_inst1.print_out_element();
  template_inst2.print_out_element();
  return 0;
}
```

실행 결과

```
첫 번째 요소: 5, 두 번째 요소: 10.5
첫 번째 요소: 문자열, 두 번째 요소: 10
```

궁금해요!

클래스는 이미 범용성을 가진 문법 아닌가요? 왜 템플릿을 사용해야 하나요?

클래스가 가진 범용성은 기능을 묶어서 추상화한 것입니다. 그런데 기능을 추상화해서 표현하더라도 데이터 형식은 고정해서 사용합니다. 반면에 템플릿을 사용하면 데이터 형식까지 범용으로 만들 수 있습니다.

클래스 템플릿에서 형식 추론

- 클래스 템플릿 객체 생성 시 **템플릿 매개변수의 형식을 명시**해야 함

```
data_package<int, double> template_inst1(5, 10.5);
data_package<string, int> template_inst2("문자열", 10);
```

  - C++ 표준 이후부터는 클래스 템플릿에서도 형식 추론 가능
  - 이전 버전의 컴파일러에서는 명시적인 데이터 형식 지정이 없으면 오류 발생

Do it! 실습 클래스 템플릿에서 형식 추론

• ch10/class_template_datatype_inference/class_template_datatype_inference.cpp

```
#include <iostream>
#include <string>

using namespace std;

template <typename Type1, typename Type2>
class data_package {
public:
  data_package(Type1 first, Type2 second) : first(first), second(second){}
  void print_out_element() {
    cout << "첫 번째 요소: " << first << ", 두 번째 요소: " << second
         << endl;
  }
private:
  Type1 first;
  Type2 second;
};

int main() {
  data_package template_inst1(5, 10.5);
  data_package template_inst2("문자열", 10);

  template_inst1.print_out_element();
  template_inst2.print_out_element();
  return 0;
}
```

실행 결과

```
첫 번째 요소: 5, 두 번째 요소: 10.5
첫 번째 요소: 문자열, 두 번째 요소: 10
```

부분 특수화

Do it! 실습 클래스 템플릿 부분 특수화

• ch10/class_template_partial_specialization/class_template_partial_specialization.cpp

```
#include <iostream>
#include <string>

using namespace std;

template <typename Type1, typename Type2>
class data_package {
public:
  data_package(Type1 first, Type2 second) : first(first), second(second){}
  void print_out_element() {
    cout << "첫 번째 요소: " << first << ", 두 번째 요소: " << second
         << endl;
  }
private:
  Type1 first;
  Type2 second;
};

template <typename T>
class data_package<string, T> {
public:
  data_package(string first, T second) : first(first), second(second){}
  void print_out_element() {
    cout << first << "과 함께 입력된" <<
         ", 두 번째 요소: " << second << endl;
  }
private:
  string first;
  T second;
};
```

부분 특수화

```
int main() {
  data_package<int, double> template_inst1(5, 10.5);
  data_package<string, int> template_inst2("문자열", 10);

  template_inst1.print_out_element();
  template_inst2.print_out_element();
  return 0;
}
```

실행 결과

```
첫 번째 요소: 5, 두 번째 요소: 10.5
문자열과 함께 입력된, 두 번째 요소: 10
```

템플릿 매개변수 형식의 일부를 특정한 형식으로 처리 할 때 부분 특수화를 사용합니다. 클래스 템플릿에서만 사용 가능하고 함수 템플릿에서는 전체 템플릿 매개변수 형식을 특정한 형식으로 처리하는 명시적 특수화만 사용 가능합니다.

- 클래스 템플릿은 ***선언과 정의가 같이*** 있어야 한다.

중첩 클래스 템플릿

- **일반 클래스와 동일하게 중첩 클래스 사용**
  - 중첩된 클래스 템플릿(**안쪽 클래스**), 기존 클래스 템플릿(**바깥쪽 클래스**)
  - 안쪽 클래스에서 바깥쪽 클래스 템플릿 매개변수를 사용하거나 새로 정의
  - 바깥쪽 클래스의 템플릿 매개변수를 사용하면 안쪽 클래스에서도 같은 데이터 형식으로 사용
- **중첩 클래스 활용법**
  - 첫 번째 : 안쪽 클래스를 **멤버 변수**로 사용
  - 두 번째 : 독립된 클래스 **객체**로 사용

Do it! 실습 중첩 클래스 템플릿

• ch10/class_template_nested_template/class_template_nested_template.cpp

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

template <typename Type1, typename Type2>
class data_package { // 바깥쪽 클래스
public:
  template <typename Type3>
 class nested_class_data_package { // 안쪽 클래스
  public:
    nested_class_data_package(Type3 data) : nested_class_data(data) {}
    Type3 get_element() { return nested_class_data; }
    Type3 nested_class_data;
  };

  template <typename Type4> // 새 템플릿 매개변수 사용
  class nested_class {      // 안쪽 클래스
  public:
    nested_class(Type4 data) : nested_class_data(data) {}
    void print_out_element() {
      cout << "중첩 클래스 데이터: " << nested_class_data << endl;
    }
  private:
   Type4 nested_class_data;
  };

  data_package(Type1 first, Type2 second) : first(first), second(second),
                                            internal_data(second){}

  void print_out_element() {
    cout << "첫 번째 요소: " << first << ", 두 번째 요소: " << second << endl;
    cout << "중첩 클래스 요소: " << internal_data.get_element() << endl;
  }
private:
  Type1 first;
  Type2 second;
  nested_class_data_package<Type2> internal_data;
};

int main() {
  data_package<string, int> template_inst1("문자열", 10);
  data_package<string, int>::nested_class<int> template_inst2(500);

  cout << "중첩 클래스 첫 번째 범례" << endl;
  template_inst1.print_out_element();

  cout << endl << "중첩 클래스 두 번째 범례" << endl;
  template_inst2.print_out_element();
  return 0;
}
```

바깥쪽 클래스의 멤버 변수를 안쪽 클래스를 사용해서 선언

독립된 객체로 선언

실행 결과

```
중첩 클래스 첫 번째 범례
첫 번째 요소: 문자열, 두 번째 요소: 10
중첩 클래스 요소: 10

중첩 클래스 두 번째 범례
중첩 클래스 데이터: 500
```

20

이지스퍼블리싱(주)

Do it! 실습 템플릿 매개변수 기본값

• ch10/class_template_default_data_type/class_template_default_data_type.cpp

```
#include <iostream>

using namespace std;

template <typename T = int>   // 기본 형식 설정
class data_package {
public:
  data_package(T first) : first(first){}
  void print_out_element() {
    cout << "템플릿 매개변수 값 : " << first << endl;
  }
private:
  T first;
};

int main() {
  data_package<> template_inst1(5);   // 기본 형식(여기서는 int)으로 지정
  data_package<string> template_inst2("클래스 템플릿 기본값이 아닌 string형");

  template_inst1.print_out_element();
  template_inst2.print_out_element();
  return 0;
}
```

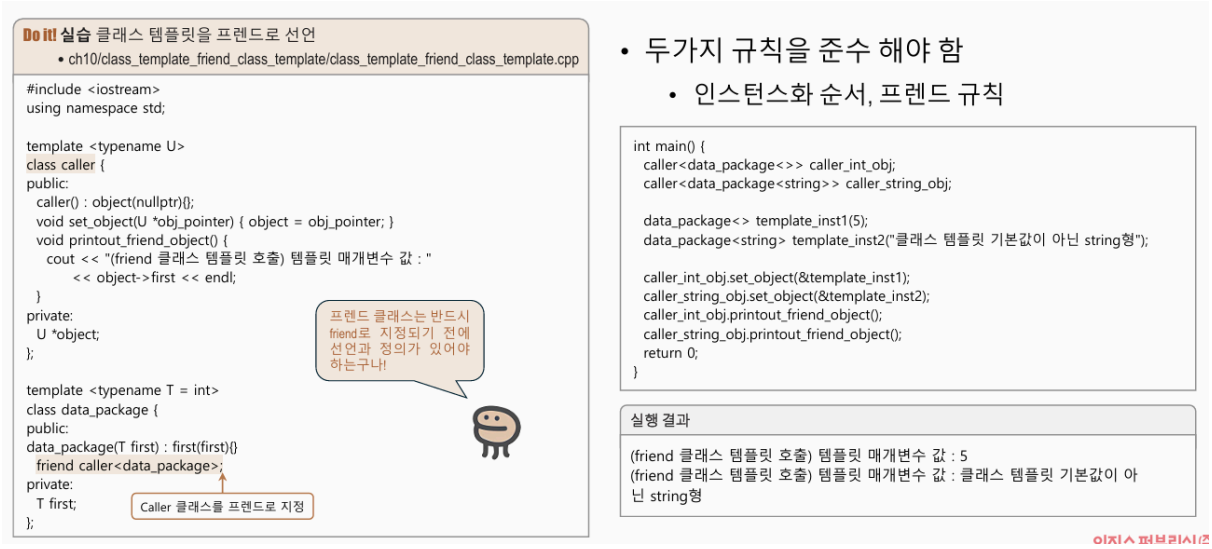
실행 결과

```
템플릿 매개변수 값 : 5
템플릿 매개변수 값 : 클래스 템플릿 기본값이 아닌 string형
```

- 일반 함수 매개변수 기본값
  - 매개변수 값을 사전에 정의
- 템플릿 매개변수 기본값
  - 템플릿 매개변수 데이터 형식을 사전에 정의

- **일반 함수 매개변수 기본값**
  - 매개변수 값을 사전에 정의
- **템플릿 매개변수 기본값**
  - 템플릿 매개변수 데이터 형식을 사전에 정의

클래스 템플릿 프렌드



- 두 가지 규칙을 준수해야 함
  - **인스턴스화 순서**, **프렌드 규칙**
- **프렌드 클래스 위치**
  - 프렌드 클래스는 **반드시 friend로 지정되기 전에 선언과 정의**가 있어야 함
  - data_package 클래스 이전에 caller 클래스 템플릿이 정의, 만약 **두 클래스 템플릿의 위치가 뒤바뀌면 오류 발생**