

10장 템플릿

- # 템플릿의 기본 개념
- # 함수 템플릿
- # 클래스 템플릿
- # 클래스 템플릿의 디폴트 템플릿 매개변수
- # 비타입 템플릿 매개변수
- # 템플릿 인스턴스화와 전문화
- # 템플릿의 동작 원리 및 주의 사항

1. 템플릿의 기본 개념

⌘ 템플릿(template)

- 프로그램을 만들어 내는 틀!?
- 어떤 타입(int, double, CPoint, ...)에 대해서도 적용이 가능한 함수 또는 클래스를 의미함

⌘ 예1 : 좌표를 나타내는 클래스

- int형 좌표, double형 좌표를 나타내는 클래스

- 기존 방법 : 별도의 클래스로 작성
 - ✓ 내용 동일

```
class CIntPoint { ... }  
class CDoublePoint { ... }  
CIntPoint P1(1, 2);  
CDoublePoint P2(1.1, 2.2);
```

- 템플릿 클래스로 작성

- ✓ 단 하나만 작성하면 됨 → 작성 방법 : 본 장의 주제

⌘ 예2 : 가변 길이 배열

- int형, double형, char형, CPoint형, ...

→ 템플릿 클래스로 만든다면 단 한번의 작성으로 처리 가능

⌘ 표준 템플릿 라이브러리(STL) : C++ 표준 라이브러리의 일부

- vector, list, stack, queue, ... → 13장

2. 함수 템플릿

다음과 같은 프로그램을 작성해 보라.

- int형 정수 2개를 더하는 함수
- double형 정수 2개를 더하는 함수
- char형 정수 2개를 더하는 함수

세 함수의 공통점과 차이점은?

* 기능 동일

* 타입(형)만 다름

→ 하나의 함수로 만드는 방법이 있을까?

→ 함수 템플릿

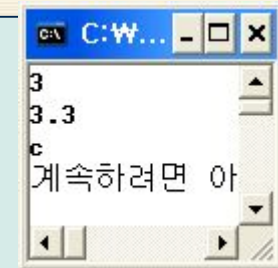
```
int Sum(int a, int b)
{
    int c = a + b;
    return c;
}
```

```
double Sum(double a, double b)
{
    double c = a + b;
    return c;
}
```

```
char Sum(char a, char b)
{
    char c = a + b;
    return c;
}
```

```
int main(void)
{
    cout << Sum(1, 2) << endl;
    cout << Sum(1.1, 2.2) << endl;
    cout << Sum('1', '2') << endl;

    return 0;
}
```



2. 함수 템플릿

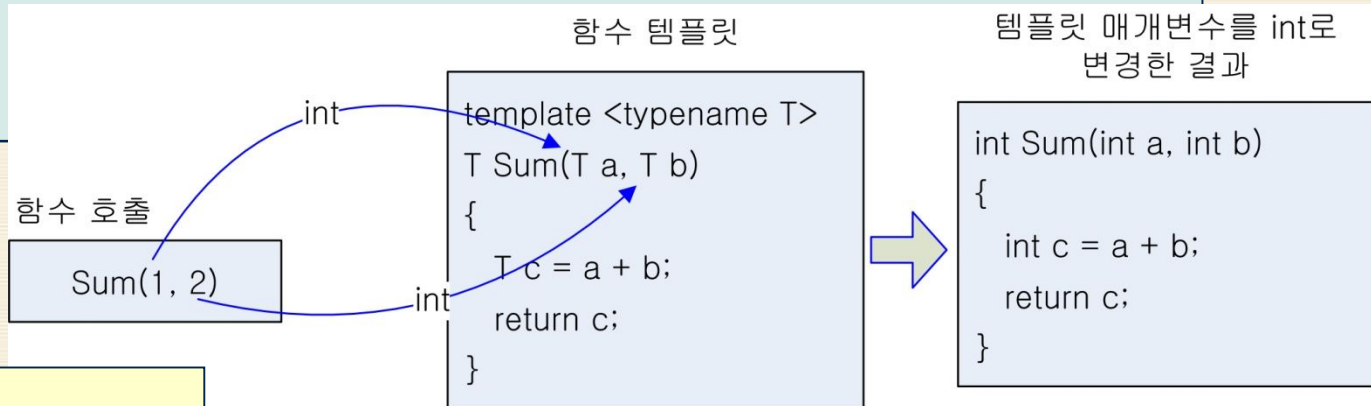
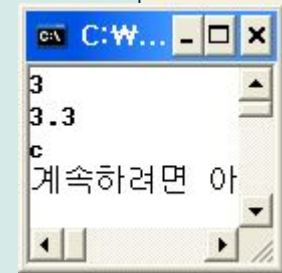
Sum 함수 템플릿

```
template <typename T> // 템플릿 선언 및 템플릿 매개변수 선언
T Sum(T a, T b)
{
    T c = a + b;
    return c;
}

int main(void)
{
    cout << Sum(1, 2) << endl; // T를 int로 대체한 함수 생성
    cout << Sum(1.1, 2.2) << endl; // T를 double로 대체한 함수 생성
    cout << Sum('1', '2') << endl; // T를 char로 대체한 함수 생성

    return 0;
}
```

typename 대신 class로 대체 가능
* 타입의 매개변수화!
→ 템플릿 매개변수



함수 템플릿
→ 실제 함수를 만드는 과정

템플릿 매개변수명(T)은
어떤 이름이든 상관없음
→ 단지 대체될 곳 통일

2. 함수 템플릿

Sum 함수 템플릿은 CPoint형 객체에 대해서도 수행 가능한가?

```
CPoint P1(1, 2), P2(3, 4);  
CPoint P3 = Sum(P1, P2);    // 수행이 가능한가?
```

- T를 CPoint로 대체해 보라.

```
CPoint Sum(CPoint a, CPoint b)  
{  
    CPoint c = a + b;  
    return c;  
}
```

- 답 : CPoint 클래스의 구현 결과에 따라 수행 여부가 결정됨
➤ + 연산자 오버로딩을 반드시 구현해야 함!

2. 함수 템플릿

Sum 함수 템플릿과 CPoint 클래스

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

```
class CPoint {
private :
    int x, y;
```

```
public :
    CPoint(int a, int b) : x(a), y(b) { }
    CPoint operator+(CPoint &Po) { return CPoint(x + Po.x, y + Po.y); }
    void Print() { cout << "(" << x << ", " << y << ")" << endl; }
};
```

```
int main(void)
{
    CPoint P1(1, 2), P2(3, 4);
    CPoint P3 = Sum(P1, P2);    // T를 CPoint로 대체한 함수 생성
    P3.Print();
    return 0;
}
```

함수 프로토타입과 함수 정의 부분 분리 구현 방법
template <typename T>까지 함수 선언 부분임.
(한 줄로 기술 가능)

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

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

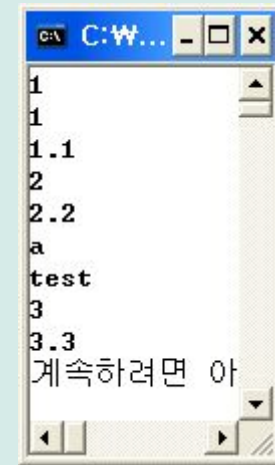

2. 함수 템플릿

2개 이상의 템플릿 매개변수 전달 가능

```
template <typename T1, typename T2, typename T3> // 3개의 템플릿 매개변수
void Print(T1 a, T2 b, T3 c)
{
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
}

int main(void)
{
    Print(1, 1, 1.1);           // (T1, T2, T3)=>(int, int, double)로 대체
    Print(2, 2.2, 'a');        // (T1, T2, T3)=>(int, double, char)로 대체
    Print("test", 3, 3.3);     // (T1, T2, T3)=>(char *, int, double)로 대체

    return 0;
}
```



다음과 같은 함수 생성

```
void Print(int a, int a, double c) { ... }
void Print(int a, double a, char c) { ... }
void Print(char *a, int a, double c) { ... }
```

3. 클래스 템플릿

▣ CIntPoint와 CDoublePoint 클래스

```
class CIntPoint {  
private :  
    int x, y;  
  
public :  
    CIntPoint(int a, int b) : x(a), y(b) { }  
    void Move(int a, int b) { x += a; y += b; }  
    void Print() { cout << "(" << x << ", " << y << ")" << endl; }  
};
```

int형 좌표 (x, y) 표현

```
class CDoublePoint {  
private :  
    double x, y;  
  
public :  
    CDoublePoint(double a, double b) : x(a), y(b) { }  
    void Move(double a, double b) { x += a; y += b; }  
    void Print() { cout << "(" << x << ", " << y << ")" << endl; }  
};
```

double형 좌표 (x, y) 표현

CIntPoint와 CDoublePoint는 타입만 제외하고 모든 내용이 동일함
→ 클래스 템플릿 사용 → 단 한번의 구현으로 OK!

```
C:\W... - [X]  
<1, 2>  
<1.1, 2.2>  
계속하려면 아
```

```
int main(void)  
{  
    CIntPoint P1(1, 2);  
    CDoublePoint P2(1.1, 2.2);  
  
    P1.Print();  
    P2.Print();  
    return 0;  
}
```


3. 클래스 템플릿

CPoint 클래스 템플릿 작성

```
template <typename T>
class CPoint {
private :
    T x, y;

public :
    CPoint(T a, T b) : x(a), y(b) { }
    void Move(T a, T b);
    void Print() { cout << "(" << x << ", " << y << ")" << endl; }
};
```

```
template <typename T>
void CPoint<T>::Move(T a, T b)
{
    x += a;
    y += b;
}
```

멤버 함수의 외부 정의 방법

객체 생성 방법
CPoint<타입명>
→ 해당 타입의 클래스를 만들어 냄!

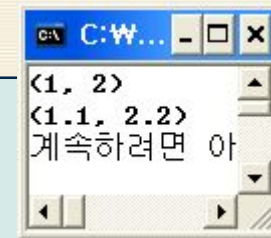
```
int main(void)
{
    CPoint<int> P1(1, 2);
    CPoint<double> P2(1.1, 2.2);

    P1.Print();
    P2.Print();

    return 0;
}
```

```
typedef CPoint<int> CIntPoint;
CIntPoint P1(1, 2);
```

// CPoint<int> P1(1, 2); 와 동일



3. 클래스 템플릿

응용 예 : 배열을 표현하는 CArray 클래스 템플릿

- 문제점 및 해결 방법은?

```
class CPoint {
private :
    int x, y;

public :
    CPoint(int a = 0, int b = 0) : x(a), y(b) {}
    void Print() { cout << "(" << x << ", " << y << " )"; }
};
```

```
template <typename T>
class CArray {
private :
    T ary[5];

public :
    CArray(T a) { for (int i = 0; i < 5; i++) ary[i] = a; }
    void Print() {
        for (int i = 0; i < 5; i++) cout << ary[i] << " ";
        cout << endl;
    }
};
```

```
int main(void)
{
    CArray<int> Ary1(5);
    CArray<CPoint> Ary2(CPoint(1, 2));

    Ary1.Print();
    Ary2.Print();

    return 0;
}
```

CPoint 객체를 원소로 갖는 배열

각 원소 출력 : CPoint 객체에 대한
<< 연산자 사용이 가능한가?
→ << 연산자 오버로딩 필요!

연산자 오버로딩을
구현한 후 수행해 보라.

3. 클래스 템플릿

CPoint 클래스를 클래스 템플릿으로 만든다면?

```
template <typename T>
class CPoint {
private :
    T x, y;

public :
    CPoint(T a = 0, T b = 0) : x(a), y(b) { }
    template <typename T>
    friend ostream &operator<<(ostream &out, CPoint<T> &Po);
};
```

friend 함수 선언

```
template <typename T> // << 연산자 오버로딩도 템플릿으로 구현
ostream &operator<<(ostream &out, CPoint<T> &Po)
{
    out << "(" << Po.x << ", " << Po.y << ")";
    return out;
}
```

3. 클래스 템플릿

코드 계속

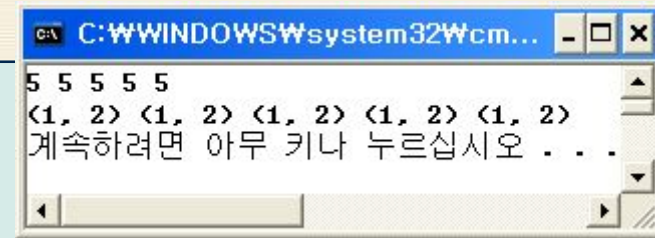
```
template <typename T>
class CArray {
private :
    T ary[5];

public :
    CArray(T a) { for (int i = 0; i < 5; i++) ary[i] = a; }
    void Print() {
        for (int i = 0; i < 5; i++) cout << ary[i] << " ";
        cout << endl;
    }
};

int main(void)
{
    CArray<int> Ary1(5);
    CArray<CPoint<int>> Ary2(CPoint<int>(1, 2)); // CPoint<int>를 원소로 가짐

    Ary1.Print();
    Ary2.Print();

    return 0;
}
```



```
C:\WINDOWS\system32\cmd.exe
5 5 5 5 5
<1, 2> <1, 2> <1, 2> <1, 2> <1, 2>
계속하려면 아무 키나 누르십시오 . . .
```

4. 클래스 템플릿의 디폴트 템플릿 매개변수

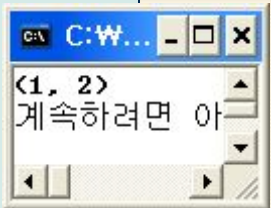
템플릿 매개변수에 대한 디폴트 타입 지정 가능

```
template <typename T = int>      // 디폴트 템플릿 매개변수
class CPoint {
private :
    T x, y;

public :
    CPoint(T a = 0, T b = 0) : x(a), y(b) { }
    void Print() { cout << "(" << x << ", " << y << ")" << endl; }
};

int main(void)
{
    CPoint<> P1(1, 2);           // CPoint<int> P1(1, 2)와 동일
    P1.Print();
}

```



디폴트 템플릿 매개변수의 디폴트 값은 뒤에서부터
지정 가능

```
template <typename T1 = int, typename T2> class CMyClass { ... }; // error
template <typename T1 = int, typename T2 = double> class CMyClass { ... }

```

5. 비타입 템플릿 매개변수

비타입 템플릿 매개변수

- int, double, CPoint와 같은 일반 타입의 템플릿 매개변수

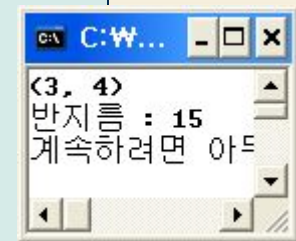
```
template <typename T, int Radius> // 비타입 템플릿 매개변수
class CPoint {
private :
    T x, y;

public :
    CPoint(T a = 0, T b = 0) : x(a), y(b) { }
    void Print() { cout << "(" << x << ", " << y << ")" << endl;
                  cout << "반지름 : " << Radius << endl; }
};

int main(void)
{
    CPoint<int, 15> P1(3, 4); // Radius로 int값 15 전달
    P1.Print();

    return 0;
}
```

비타입 템플릿 매개변수는
r-value로만(읽기) 사용 가능



5. 비타입 템플릿 매개변수

비타입 템플릿 매개변수로 올 수 있는 타입

- 정수형 상수 또는 정수형 상수 표현식
- 전역 변수 또는 전역 객체의 주소 : & 주소 연산자를 사용하여 전달하며 배열과 함수의 경우 & 생략 가능, 템플릿 형식매개변수는 포인터로 받음
- 전역 변수 또는 전역 객체 : 템플릿 형식매개변수는 참조로 받음

예

```
template <typename T, int Radius> CPoint { };  
int a = 5; CPoint<int, a> P1; // X, 상수가 아님  
template <typename T, char *p> CPoint { };  
CPoint<int, "C++"> P1; // X, 정수형 상수가 아님  
char p[] = "C++"; CPoint<int, p> P1; // O, 주소 전달 가능  
template <typename T, double p> CPoint { }; // X, 정수형 매개변수가 아님  
template <typename T, int &Radius> CPoint { };  
int a = 5; CPoint<int, a> P1; // O, 참조 전달 가능
```

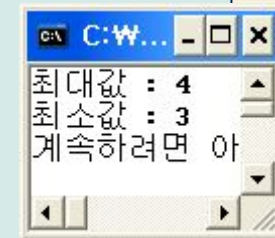
5. 비타입 템플릿 매개변수

함수 템플릿에서도 비타입 템플릿 매개변수 사용 가능

```
template <typename T, bool x> // 비타입 템플릿 매개변수 bool x
T MaxMin(T a, T b)
{
    if (x == true)
        return ((a > b) ? a : b);
    else
        return ((a < b) ? a : b);
}

int main(void)
{
    cout << "최대값: " << MaxMin<int, true>(3, 4) << endl;
    cout << "최소값: " << MaxMin<int, false>(3, 4) << endl;

    return 0;
}
```



함수 템플릿에서도 함수 사용 시
템플릿 매개변수에 대한 타입 명기 가능

6. 템플릿 인스턴스화와 전문화

⌘ 템플릿 인스턴스화 (template instantiation)

- 함수 템플릿이나 클래스 템플릿으로부터 특정 타입의 함수 및 클래스를 만들어 내는 과정
- 묵시적 인스턴스화 : 컴파일러에 의해 자동으로 생성, 지금까지의 예
- 명시적 인스턴스화 : 프로그래머가 특정 타입에 대한 함수 및 클래스를 만들어내도록 명시적으로 컴파일러에게 요청

⌘ 템플릿 전문화 (template specialization)

- 템플릿 인스턴스화를 통해 만들어진 함수 및 클래스와 명시적 전문화를 통해 만들어진 함수 및 클래스를 모두 지칭하는 말
- 명시적 전문화 : 특정 타입에 대해서 기존 템플릿과는 다른 내용으로 함수 또는 클래스를 만들 수 있음

6. 템플릿 인스턴스와 전문화

명시적 인스턴스의 예

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

컴파일러에 의해 **int**형에 대한 함수가 생성됨
내용은 템플릿과 동일함

```
template int Sum(int a, int b); // int형 Sum 함수의 명시적 인스턴스화
```

```
int main(void)
{
```

더 이상 묵시적 인스턴스화는 수행되지 않음

```
    int x = Sum(3, 4);
```

```
    double y = Sum(1.1, 2.2);
```

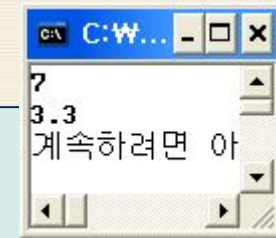
묵시적 인스턴스화가 수행됨

```
    cout << x << endl;
    cout << y << endl;
```

```
    return 0;
```

```
}
```

클래스 템플릿에 대한 명시적 인스턴스화의 예
emplate <typename T> class CPoint { ... };
template class CPoint<int>;



6. 템플릿 인스턴스화와 전문화

명시적 전문화의 예

- Sum 함수에 대해 int형의 경우 다른 의미를 부여하고자 함

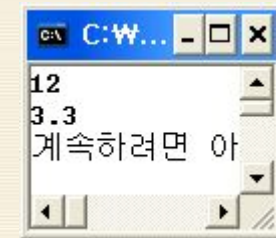
```
template <typename T>
T Sum(T a, T b)
{
    T c = a + b;
    return c;
}
```

```
template <>
int Sum(int a, int b)
{
    int c = a * b;
    return c;
}
```

```
int main(void)
{
    int x = Sum(3, 4);
    double y = Sum(1.1, 2.2);

    cout << x << endl;
    cout << y << endl;

    return 0;
}
```



int형 Sum 함수에 대한 명시적 전문화
기존 템플릿과는 내용이 다름!
문법 : `template<>` 다음에 함수가 옴

일반 함수 `Sum(int, int)`가 존재한다면
수행 우선 순위는?

일반 함수 → 명시적 전문화 → 템플릿

클래스 템플릿에 대한 명시적 전문화의 예
`template <typename T> class CPoint { ... };`
`template <> CPoint<int> { ... };`

7. 템플릿의 동작 원리 및 주의 사항

⌘ 함수(또는 클래스)를 사용하기 위해서는

- 컴파일 시 : 함수의 프로토타입 이상을 알아야만 됨
- 링크 시 : 프로그램 어딘가에 함수의 정의가 단 한번 나와야 됨

⌘ 템플릿

- 함수나 클래스를 만들어낼 수 있는 틀을 제공
- 컴파일 시 : 실제 함수 또는 클래스를 만들어 냄
 - 컴파일 : 파일 단위로 수행됨
- 컴파일 시 템플릿 인스턴스화를 수행할 때 실제 함수 몸체까지 만들어야 하므로 템플릿의 선언과 정의 부분을 해당 파일 내에 포함하고 있어야만 함

⌘ 일반적인 템플릿의 사용 방법

- 템플릿을 사용하는 파일 내에 템플릿 선언과 정의를 모두 포함 → 지금까지의 방법
- 헤더 파일에 템플릿 선언과 정의를 모두 포함하고 이를 include하여 사용 → 보다 일반적인 방법!

7. 템플릿의 동작 원리 및 주의 사항

template.h

```
template <typename T> // 함수 템플릿은 함수 몸체 포함
T Sum(T a, T b)
{
    T c = a + b;
    return c;
}

void func(int a); // 일반 함수는 프로토타입만
```

template.cpp

```
#include <iostream>
#include "template.h"
using namespace std;

void func(int a)
{
    cout << Sum(a, a) << endl;
}
```

main.cpp

```
#include <iostream>
#include "template.h"
using namespace std;

int main(void)
{
    cout << Sum(1, 2) << endl;
    func(3);

    return 0;
}
```