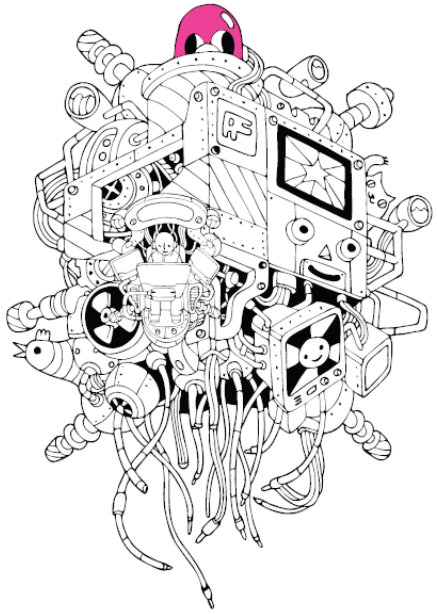
윤성우의
열혈 C++ 프로그래밍



윤성우 저 열혈강의 C++ 프로그래밍 개정판

Chapter 13. 템플릿(Template) 1

윤성우의

열혈 C++ 프로그래밍

Chapter 13-1. 템플릿(Template)에 대한 이해와 함수 템플릿

윤성우 저 열혈강의 C++ 프로그래밍 개정판

함수를 대상으로 템플릿 이해하기


```
int Add(int num1, int num2)
{
    return num1+num2;
}
```

일반함수

```
T Add(T num1, T num2)
{
    return num1+num2;
}
```

템플릿화의 중간 단계

```
template <typename T>
T Add(T num1, T num2)
{
    return num1+num2;
}
```

템플릿화 완료

함수를 찍어내는 틀

```
template <typename T>
T Add(T num1, T num2)
{
    return num1+num2;
}
int main(void)
{
    cout<< Add<int>(15, 20)         <<endl;
    cout<< Add<double>(2.9, 3.7)    <<endl;
    cout<< Add<int>(3.2, 3.2)       <<endl;
    cout<< Add<double>(3.14, 2.75)  <<endl;
    return 0;
}
```

T를 int로 하여 만들어진 함수를 호출

```
int Add(int num1, int num2)
{
    return num1+num2;
}
```

T를 double로 하여 만들어진 함수를 호출

```
double Add(double num1, double num2)
{
    return num1+num2;
}
```

컴파일러가 생성하는 템플릿 기반의 함수


```
int Add<int>(int num1, int num2)
{
     return num1+num2;
}
```

Add<int>(…) 의 함수호출 문을 처음 컴파일할때 이 함수가 만들어진다.

```
double Add<double>(double num1, double num2)
{
     return num1+num2;
}
```

Add<double>(…) 의 함수호출 문을 처음 컴파일할때 이 함수가 만들어진다.

호출하기가 좀 불편한 건 있네요.

→ 되도록 쓰는게 좋다

```
template <typename T>
T Add(T num1, T num2)
{
    return num1+num2;
}
```

호출하기 불편하지 않다! 컴파일러가 전달인자의 자료형을 통해서 호출해야 할 함수의 유형을 자동으로 결정해주기 때문이다.

```
int main(void)
{
    cout<< Add(15, 20)      <<endl;
    cout<< Add(2.9, 3.7)    <<endl;
    cout<< Add(3.2, 3.2)    <<endl;
    cout<< Add(3.14, 2.75)  <<endl;
    return 0;
}
```

전달되는 인자를 통해서 컴파일러는 이를 다음과 같이 해석한다.

Add<int>(15, 20);

전달되는 인자를 통해서 컴파일러는 이를 다음과 같이 해석한다.

Add<double>(2.9, 3.7)

함수 템플릿과 템플릿 함수


```
template <typename T>
T Add(T num1, T num2)
{
    return num1+num2;
}
```

함수의 형태로 정의된 템플릿이기 때문에 함수 템플릿이라 한다. 즉, 이는 템플릿이지 호출이 가능한 형태의 함수가 아니다.

```
int Add<int>(int num1, int num2)
{
    return num1+num2;
}

double Add<double>(double num1, double num2)
{
    return num1+num2;
}
```

이는 템플릿 함수이다. 템플릿을 기반으로 컴파일러에 의해서 생성된 함수이기 때문이다. 즉, 이는 함수이지 템플릿이 아니다.

둘 이상의 형(Type)에 대해 템플릿 선언하기

도대체 이런 typename 사용

```
template <class T1, class T2>
void ShowData(double num)
{
    cout<<(T1)num<<", "<<(T2)num<<endl;
}

int main(void)
{
    ShowData<char, int>(65);
    ShowData<char, int>(67);
    ShowData<char, double>(68.9);
    ShowData<short, double>(69.2);
    ShowData<short, double>(70.4);
    return 0;
}
```

이런 콤마

이렇듯 콤마를 이용해서 둘 이상의 형에 대해서 템플릿을 선언할 수 있다.

실행결과

```
A, 65
C, 67
D, 68.9
69, 69.2
70, 70.4
```

템플릿의 선언에 있어서 키워드 typename과 class는 같은 의미로 사용된다.

함수 템플릿의 특수화(Specialization): 도입


```
template <typename T>
T Max(T a, T b)
{
    return a > b ? a : b ;
}

int main(void)
{
    cout<< Max(11, 15)              <<endl;
    cout<< Max('T', 'Q')            <<endl;
    cout<< Max(3.5, 7.5)            <<endl;
    cout<< Max("Simple", "Best")    <<endl;
    return 0;
}
```

대소비교 함수 템플릿! 큰 값을 반환한다.

정수, 실수 그리고 문자를 대상으로는 Max 함수의 호출의 의미를 갖는다. 그러나 문자열을 대상으로 호출이 되면 의미를 갖지 않는다!

대소비교가능

```
const char* Max(const char* a, const char* b)
{
    return strlen(a) > strlen(b) ? a : b ;
}
```

문자열의 길이비교가 목적인 경우 어울리는 함수!

문자열함수 오버로딩

일반적인 상황에서는 Max 템플릿 함수가 호출되고, 문자열이 전달되는 경우에는 문자열의 길이를 비교하는 Max 함수를 호출하게 할 수 없을까? → 함수 템플릿의 특수화 등장 배경

함수 템플릿의 특수화(Specialization): 적용


```
template <typename T>
T Max(T a, T b)
{
    return a > b ? a : b ;
}
template <>
char* Max(char* a, char* b)
{
    cout<<"char* Max<char*>(char* a, char* b)"<<endl;
    return strlen(a) > strlen(b) ? a : b ;
}
template <>
const char* Max(const char* a, const char* b)
{
    cout<<"const char* Max<const char*>(const char* a, const char* b)"<<endl;
    return strcmp(a, b) > 0 ? a : b ;
}
```

함수 템플릿 Max를
char * 형에 대해서 특수화

함수 템플릿 Max를
const char * 형에 대해서 특수화

```
int main(void)
{
    cout<< Max(11, 15)              <<endl;
    cout<< Max('T', 'Q')            <<endl;
    cout<< Max(3.5, 7.5)            <<endl;
    cout<< Max("Simple", "Best")    <<endl;
    char str1[]="Simple";
    char str2[]="Best";
    cout<< Max(str1, str2)          <<endl;
    return 0;
}
```

실행결과

```
15
T
7.5
const char* Max<const char*>(const char* a, const char* b)
Simple
char* Max<char*>(char* a, char* b)
Simple
```

함수 템플릿의 특수화(Specialization): 비교


```
template <>
char* Max(char* a, char* b)
{ . . . . }

template <>
const char* Max(const char* a, const char* b)
{ . . . . }
```

특수화하는 자료형의 정보가 생략된 형태

```
template <>
char* Max<char *>(char* a, char* b)
{ . . . . }

template <>
const char* Max<const char *>(const char* a, const char* b)
{ . . . . }
```

특수화하는 자료형의 정보를 명시한 형태

윤성우의

열혈 C++ 프로그래밍

Chapter 13-2. 클래스 템플릿(Class Template)

윤성우 저 열혈강의 C++ 프로그래밍 개정판

클래스 템플릿의 정의


```
class Point
{
private:
    int xpos, ypos;
public:
    Point(int x=0, int y=0) : xpos(x), ypos(y)
    { }
    void ShowPosition() const
    {
        cout<<'['<<xpos<<", "<<ypos<<']'<<endl;
    }
};
```

일반 클래스

```
template <typename T>
class Point
{
private:
    T xpos, ypos;
public:
    Point(T x=0, T y=0) : xpos(x), ypos(y)
    { }
    void ShowPosition() const
    {
        cout<<'['<<xpos<<", "<<ypos<<']'<<endl;
    }
};
```

클래스의 템플릿화

```
int main(void)
{
    Point<int> pos1(3, 4);          // 템플릿 클래스 Point<int> 형 객체 생성
    pos1.ShowPosition();

    Point<double> pos2(2.4, 3.6);   // 템플릿 클래스 Point<double> 형 객체 생성
    pos2.ShowPosition();

    Point<char> pos3('P', 'F');     // 템플릿 클래스 Point<char> 형 객체 생성
    pos3.ShowPosition();
    return 0;
}
```

템플릿 클래스의 객체생성시 자료형의 정보는 생략이 불가능하다!

클래스 템플릿의 선언과 정의의 분리


```
template <typename T>
class SimpleTemplate
{
public:
    T SimpleFunc(const T& ref);
};
```

함수의 선언

```
template <typename T>
T SimpleTemplate<T>::SimpleFunc(const T& ref)
{
    . . . .
}
```

템플릿 외부의 함수정의

SimpleTemplate :: SimpeFunc(. . . .)

√ SimpleTemplate 클래스의 멤버함수 SimpleFunc를 의미함

SimpleTemplate<T> :: SimpleFunc(. . . .)

√ T에 대해서 템플릿으로 정의된 SimpleTemplate의 멤버함수 SimpleFunc를 의미함

template <typename T>

√ <T>가 들어가면 이 T가 의미하는 바를 항상 설명해야 한다.

헤더파일과 소스파일의 구분


```
#ifndef _ _POINT_TEMPLATE_H_
#define _ _POINT_TEMPLATE_H_

template <typename T>
class Point
{
private:
    T xpos, ypos;
public:
    Point(T x=0, T y=0);
    void ShowPosition() const;
 };
#endif
```

헤더파일

```
#include <iostream>
#include "PointTemplate.h“
using namespace std;
int main(void)
{
    Point<int> pos1(3, 4);
    pos1.ShowPosition();
    Point<double> pos2(2.4, 3.6);
    pos2.ShowPosition();
    Point<char> pos3('P', 'F');
    pos3.ShowPosition();
    return 0;
}
```

소스파일 1

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

template <typename T>
Point<T>::Point(T x, T y) : xpos(x), ypos(y) {   }

template <typename T>
void Point<T>::ShowPosition() const
{
    cout<<'['<<xpos<<", "<<ypos<<']'<<endl;
}
```

소스파일 2

기준에 의해서 헤더파일과 소스파일을 잘 구분하였다.

그러나 컴파일 오류가 발생한다! 이유는?

파일을 나눌 때에는 고려할 사항이 있습니다.


```
#ifndef _ _POINT_TEMPLATE_H_
#define _ _POINT_TEMPLATE_H_

template <typename T>
class Point
{
private:
    T xpos, ypos;
public:
    Point(T x=0, T y=0);
    void ShowPosition() const;
 };
#endif
```

헤더파일

```
#include <iostream>
#include "PointTemplate.h“
using namespace std;
int main(void)
{
    Point<int> pos1(3, 4);
    pos1.ShowPosition();
    Point<double> pos2(2.4, 3.6);
    pos2.ShowPosition();
    Point<char> pos3('P', 'F');
    pos3.ShowPosition();
    return 0;
}
```

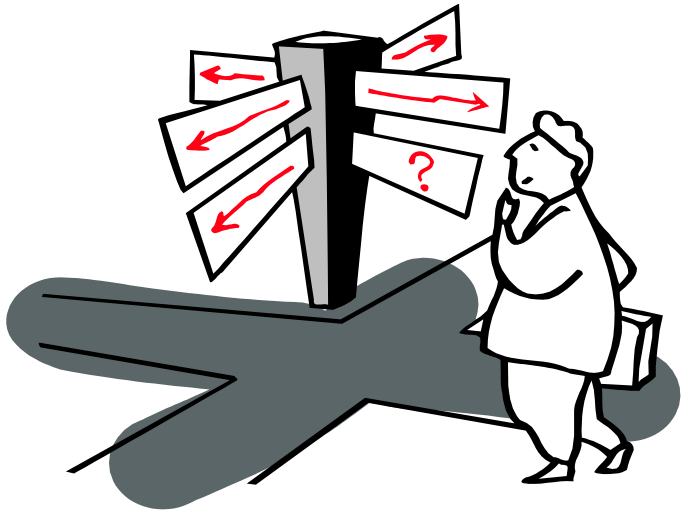
소스파일 1

위의 소스파일을 컴파일 할 때 Point<int>, Point<double>, Point<char> 템플릿 클래스가 만들어져야 한다. 따라서 Point 클래스 템플릿의 정의 부에 대한 정보도 필요로 한다.

```
#include <iostream>
#include "PointTemplate.h"
#include "PointTemplate.cpp"
using namespace std;
int main(void)
{
    . . . .
    return 0;
}
```

해결책 1. 클래스 템플릿의 정의 부를 담고 있는 소스파일을 포함시킨다.

해결책 2. 헤더파일에 클래스 템플릿의 정의 부를 포함시킨다.



Chapter 13이 끝났습니다. 질문 있으신지요?