C++ 프로그래밍

강의노트 #11: 템플릿 (Template)

2007. 5. 30.

담당교수: 조 재 수

E-mail: jaesoo27@kut.ac.kr

학습내용

- 템플릿에 대한 이해
- 함수 템플릿
- 클래스 템플릿
- 실습문제

1. 템플릿을 사용하는 이유

- 프로그램을 작성할 때 가장 중요하게 생각하는 문제 -> 프로그램의 각 요소를 일반화하여 최대한 많은 곳에서 이미 작성된 코드를 재사용
  예 1) 함수의 구동 방식을 일반화하고, 사용하는 타입에 대한 제한을 없애기 위해 void*를 사용
  예 2) 가상함수나 오버로딩을 이용해 다형성을 제공
- C++ 언어에서는 이미 작성된 프로그램을 재사용할 수 있도록 많은 고민을 해왔고, 그 결과로 프로그램의 일반화(generalization)에 초점을 맞춘 템플릿(template)이라는 해결책
- 템플릿 방식은 가상함수를 이용한 것이나, 오버로딩과 같은 방법과는 전혀 다른 형태의 다형성을 제공

1. 템플릿을 사용하는 이유

- 템플릿은 C++언어의 다른 기술들과 달리 객체지향과는 크게 관계없이 오직 일반화에만 초점을 맞추어 코드의 재사용성을 극대화할 수 있도록 한다는 데 큰 특징
- **템플릿을 사용하는 이유는 템플릿을 사용하면 적은 노력으로 훌륭한 코드를 얻을 수 있기 때문, 또는 컴퓨터에게 필요한 정보만 전달해 주면 컴퓨터가 다양한 클래스와 함수를 만들어 준다.**

1. 템플릿을 사용하는 이유

예)

```
int max(int a, int b)
{
  return (a > b ? a : b);
}
```

- 매개변수가 float 타입의 두 값을 비교하고, 싶을 때는 어떻게 할까? float 타입의 인자를 받는 오버로딩된 다음과 같은 max() 함수를 만들어 주어야 하는 것일까?

```
float max(float a, float b)
{
    return (a > b ? a : b);
}
```

- 매개변수가 double 이면? 해결책 -> 템플릿 함수 선언

1. 템플릿을 사용하는 이유

- **함수 템플릿 선언**

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

template <typename T> 는 "**T라는 이름(type name)에 대해서, 다음에 정의하는 대상을 템플릿으로 선언한다.**" 라는 것을 컴파일러에게 알려주는 역할을 한다.

- 참고로 T라는 문자 대신 다른 문자를 사용해도 상관없다.

1. 템플릿(template)에 대한 이해

- 템플릿: 모형자라는 뜻으로 기능은 결정되어 있고, 색(color)은 결정되어 있지 않은 특징

```
int Add(int a, int b)
{
    return a+b;
}
```

템플릿화

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

- 템플릿에서 T는 자료형을 결정짓지 않겠다는 의미

1. 템플릿(template)에 대한 이해

```
#include <iostream>
using std::endl;
using std::cout;

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

int main(void)
{
    cout<<Add(10, 20)<<endl;
    cout<<Add(1.1, 2.2)<<endl;

    return 0;
}
```

- Template <typename T>의 의미
  - T라는 이름(typename)에 대해서, 다음에 정의하는 대상을 템플릿으로 선언한다.
- Template <typename T> ==

  Template <class T> ==

  Template <typename K> 모두 같은 의미

1. 템플릿(template)에 대한 이해

```
/* 예제 12-1.cpp */
1: #include <iostream>
2: using std::endl;
   using std::cout;
3:
4: template <typename T>
5: T max(T a, T b)
6: {
7:    return (a > b ? a : b);
8: }
9:
10: int main(void)
11: {
12:   cout<< max(10, 20)<<endl;
13:   cout<< max(1.1, 2.2)<<endl;
14:
15:   return 0;
16: }
```

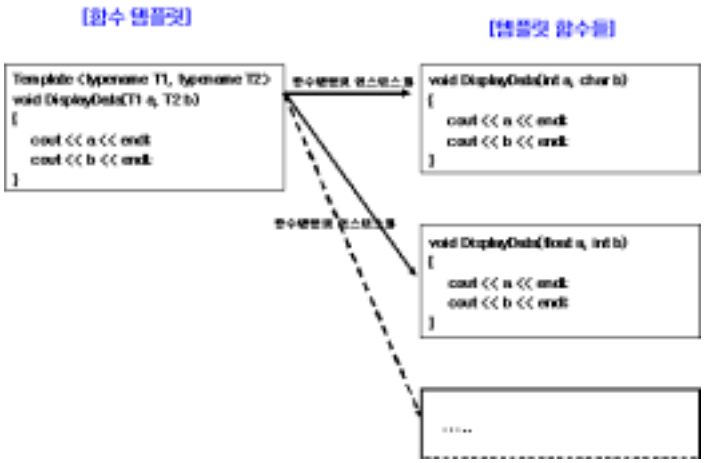

2. 함수 템플릿-[1] 함수 템플릿의 원리

```
/* 예제 12-2.cpp */
1: #include <iostream>
2: using namespace std;
3:
4: template <typename T1, typename T2> // 함수 템플릿 정의
5: void DisplayData(T1 a, T2 b)
6: {
7:    cout<< a <<endl;
8:    cout<< b <<endl;
9: }
10:
11: int main(void)
12: {
13:   int i= 4;
14:   char c = 'A';
15:
16:   DisplayData(i, c);
17:
18:   float f = 2.5;
19:   DisplayData(f, i);
20:
21:   int j = 10;
22:   char d = 'B';
23:   DisplayData(j,d);
24:
25:   return 0;
26: }
```

실행결과:

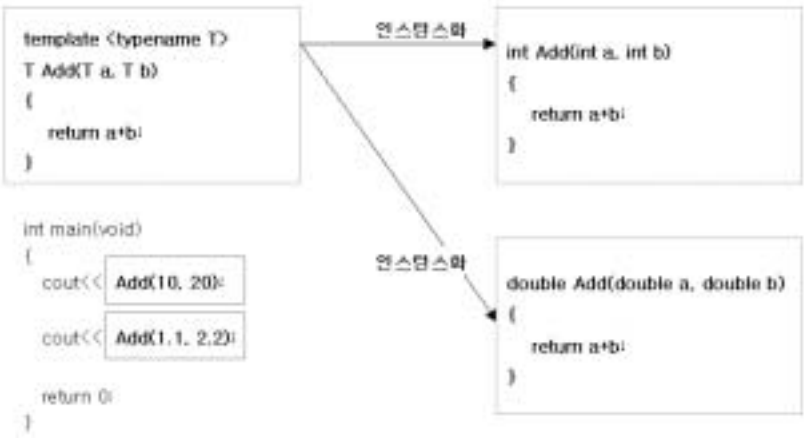
```
4
A
2.5
4
10
B
```

2. 함수 템플릿 - [1] 함수 템플릿의 원리



[그림 12-1] 함수 템플릿의 인스턴스화

<템플릿의 원리 이해>



4. 템플릿의 원리 이해

- 지금까지 정의한 함수 템플릿들은 그 자체가 호출이 가능한 함수가 아니다. 다만 호출이 가능한 함수를 만들기 위한 하나의 틀이다.
- 함수 템플릿을 기반으로 해서 만들어지는(컴파일러에 의하여), 실제 호출이 가능한 함수들을 가리켜 **템플릿 함수라고** 부른다.
- 템플릿 함수가 생성되는 현상을 가리켜 **"함수 템플릿의 인스턴스화"** 라고 한다.
- 클래스 템플릿은 객체화가 가능한 클래스를 만들기 위한 하나의 틀
- 템플릿 클래스는 선언과 정의를 각각 다른 파일(하나는 .h, 또 하나는 .cpp)에 분리시켜 놓을 수 없다. 이유는?
- 클래스 템플릿은 하나의 파일 내에 선언과 정의가 함께 있어야 컴파일과정에서 에러가 없다.

2. 함수 템플릿

- 함수 템플릿 vs. 템플릿 함수
- 둘 이상의 타입에 대해서 템플릿화
  - IntroTemplate3.cpp
- 함수 템플릿의 특수화
  - SpeciFuncTemplate2.cpp

```
template <typename T>
int SizeOf(T a)
{
    return sizeof(a);
}
```

특수화 →

```
template<>
int SizeOf(char* a)
{
    return strlen(a);
}
```

2. 함수 템플릿 –[2] 함수 템플릿의 특수화

```
// SpeciFuncTemplate1.cpp
#include <iostream>
using std::endl;
using std::cout;

template <typename T> // 함수 템플릿 정의

int SizeOf(T a)
{
      return sizeof(a);
}

int main(void)
{
      int i=10;
      double e=7.7;
      char* str="한국기술교육대학교 최고!";

      cout<<SizeOf(i)<<endl;
      cout<<SizeOf(e)<<endl;
      cout<<SizeOf(str)<<endl;

      return 0;
}
```

▪Sizeof(str)에서는 전달되는 인자가 문자열을 가리키는 포인터일 경우, 문자열의 길이를 리턴해 주기를 원한다면 어떻게 하면 될까?

▪다음과 같이 반환하면 될 것이다.

return strlen(str); // str이 문자열 포인터인 경우

▪**템플릿 특수화**라는 문법적 요소를 사용

2. 함수 템플릿 – 템플릿 특수화 예제

```
/*   SpeciFuncTemplate2.cpp   */
#include <iostream>
using std::endl;
using std::cout;

template <typename T> // 함수 템플릿 정의
int SizeOf(T a)
{
      return sizeof(a);
}

template<> // 함수 템플릿  특수화 선언
int SizeOf(char* a)  // 전달인자가 char*인 경우에는 이 함수를 호출
{
      return strlen(a);
}

int main(void)
{
      int i=10;
      double e=7.7;
      char* str=" 한국기술교육대학교 최고 !";

      cout<<SizeOf(i)<<endl;
      cout<<SizeOf(e)<<endl;
      cout<<SizeOf(str)<<endl;

      return 0;
}
```

- SpeciFuncTemplate1.cpp의 경우 전달되는 인자가 문자열을 가리키는 포인터일 경우, 문자열의 길이를 리턴해 주기를 원한다면, 즉

  return strlen(a);

  -> 템플릿 특수화라는 문법적 요소를 사용해서 해결

- Template<> int SizeOf(char* a)
- Template<> int SizeOf<>(char* a)
- Template<> int SizeOf<char*>(char* a)

  -> 모두 같은 표현

3. 클래스 템플릿

```
/* 예제 12-5.cpp */
#include <iostream>
using namespace std;


class Data
{
      int data;
public:
      Data(int d){
            data=d;
      }
      void SetData(int d){
            data=d;
      }
      int GetData(){
            return data;
      }
};

int main(void)
{
      Data d1(0);
      d1.SetData(10);
      Data d2(100);

      cout<<d1.GetData()<<endl;
      cout<<d2.GetData()<<endl;

      return 0;
}
```

• Data 클래스의 자료형을 int형뿐만 아니라, float, char 형으로도 사용할 수 있도록 클래스를 템플릿화

3. 클래스 템플릿

```
class Data
{
   int data;
public:
   Data(int d){  data=d; }
   void SetData(int d){
      data=d;
   }
   int GetData(){
      return data;
   }
};
```

템플릿화 →

```
template <typename T>
class Data
{
   T data;
public:
   Data(T d){ data=d; }
   void SetData(T d){
      data=d;
   }
   T GetData(){
      return data;
   }
};
```

```
int main(void)
{
Data<int> d1(0); // T를 int로 간주하고 객체생성
d1.SetData(10);
Data<char> d2('a'); // T를 char로 간주하고 객체생성
cout << d1.GetData() << endl;
cout << d2.GetData() << endl;
return 0;
}
```

클래스 템플릿을 기반으로 객체 생성시 결정하고자 하는 자료형을 명시적으로 선언해 주어야 하는 이유는 무엇인가?

[힌트] 객체 생성과정을 생각해보자.

3. 클래스 템플릿

```
template <typename T>
class Data
{
    T data;
public:
    Data(T d){ data=d; }
    void SetData(T d){
        data=d;
    }
    T GetData(){
        return data;
    }
};
```

선언/정의 분리 가능 →

```
template <typename T>
class Data
{
    T data;
public:
    Data(T d);
    void SetData(T d);
    T GetData();
};

template <typename T>
Data<T>::Data(T d){
    data=d;
}
template <typename T>
void Data<T>::SetData(T d){
    data=d;
}
template <typename T>
T Data<T>::GetData(){
    return data;
}
```


3. 클래스 템플릿 – 스택 클래스의 템플릿화

```
/*   StackTemplate1.cpp  */

#include <iostream>
using std::cout;
using std::endl;

class stack {
private:
    int topIdx;   // 마지막 입력된 위치의 인덱스.
   char* stackPtr;  // 스택 포인터.
public:
      stack(int s=10);
      ~stack();
      void Push(const char& pushValue);
      char Pop();
};

stack::stack(int len){
      topIdx=-1;          // 스택 인덱스 초기화.
      stackPtr=new char[len];// 데이터 저장 위한
      배열 선언.
}

stack::~stack(){
      delete[] stackPtr;
}
```

```
void stack::Push(const char& pushValue){
// 스택에 데이터 입력.
      stackPtr[++topIdx]=pushValue;
}
char stack::Pop(){ // 스택에서 데이터 꺼냄.
      return stackPtr[topIdx--];
}

int main()
{
      stack stack(10);
      stack.Push('A');
      stack.Push('B');
      stack.Push('C');

      for(int i=0; i<3; i++){
            cout<<stack.Pop()<<endl;
      }

      return 0;
}
```

```
/*   StackTemplate2.cpp */

#include <iostream>
using std::cout;
using std::endl;

template <typename T>
class stack {
private:
   int topIdx;   // 마지막 입력된 위치의 인덱스.
  T* stackPtr;  // 스택 포인터.
public:
       stack(int s=10);
       ~stack();
       void Push(const T& pushValue);
       T Pop();
};

template <typename T>
stack<T>::stack(int len){
       topIdx=-1;                // 스택 인덱스 초기화.
       stackPtr=new T[len];   // 데이터 저장 위한 배열 선언.
}

template <typename T>
stack<T>::~stack(){
       delete[] stackPtr;
}

template <typename T>
void stack<T>::Push(const T& pushValue){ // 스택에 데
       이터 입력.
       stackPtr[++topIdx]=pushValue;
}

template <typename T>
T stack<T>::Pop(){ // 스택에서 데이터 꺼냄.
       return stackPtr[topIdx--];
}

 int main(){
       stack<char> stack1(10);
       stack1.Push('A');
       stack1.Push('B');
       stack1.Push('C');

    for(int i=0; i<3; i++){
                cout<<stack1.Pop()<<endl;
       }

       stack<int> stack2(10);
       stack2.Push(10);
       stack2.Push(20);
       stack2.Push(30);

       for(int j=0; j<3; j++){
                cout<<stack2.Pop()<<endl;
       }
       return 0;
}
```


5. 표준 템플릿 라이브러리(STL)

- 표준 템플릿 라이브러리(Standard Template Library)는 템플릿을 사용해서 만들어진 표준화된 라이브러리
- 라이브러리란 클래스나 함수를 모아놓은 것을 말한다.
- STL은 일반적으로 많이 사용하는 클래스와 함수들이 만들어져 있는데, 예를 들어 링크드 리스트 클래스, 동적 배열 클래스, 정렬 함수, 검색 함수 등과 같이 범용적인 클래스와 함수들이 있다.
- 또한 이런 클래스와 함수들은 템플릿으로 만들어져 있기 때문에 확장이 용이하다.

5. 표준 템플릿 라이브러리(STL)

- STL을 사용함으로써 얻을 수 있는 장점
  - 첫번째로 STL은 표준이다. 이는 다른 프로그래머도 우리와 똑같은 클래스와 함수를 사용한다는 것을 의미한다. 그래서 다른 사람이 작성해 놓은 코드를 쉽게 알아볼 수 있다.
  - STL은 전문가들이 만들어 놓은 것이기 때문에 여러분이 직접 만든 것보다 효율적이고 안전하다.

6. 실습 및 Homework 문제 1

1. 다음의 Point 클래스를 이용하여, 이 Point 객체를 Add 함수 템플릿의 인자로 전달하려고 한다. 가능한가? 만약 불가능하다면 Point 클래스를 적절히 변경하세요. **[힌트] Point 클래스끼리의 + 연산이 기본적으로 가능하도록 먼저 클래스를 정의하는 것이 필요하다.**

```
class Point{
  private:
    int x, y;
  public:
       Point(int _x=0, int _y=0) : x(_x), y(_y){}
     void ShowPosition();
};
void Point::ShowPosition(){
cout << x << “  “ << y << endl;
}
```

```
int main(void)
{
   Point p1(1,2);
   Point p2(1,2);

  Point p3 =Add(p1, p2);
  p3.ShowPosition();

   return 0;
}

실행 예
2  4
```

5. 실습 및 Homework 문제 2

2. 전달되는 두 개의 변수 값을 변경시켜 주는 Swap 함수를 정의해 보자. 물론 template로 구현을 해서 어떠한 자료형의 변수이건 변경이 가능하게끔 구현해야 한다. 문제 1에서 제시하고 있는 Point 객체까지도 변경가능해야 한다.

```
Main 함수의 예
int main(void)
{
   Point p1(1,2);
   Point p2(100,200);

   Swap(p1, p2);
   p1.ShowPosition();  // 100, 200 출력
   p2.ShowPosition();   // 1, 2 출력

  int a = 10, b = 20;
  Swap(a, b);
  cout << a << ' ' << b << 두이;  // 20 , 10 출력
   return 0;
}
```


질문 & 답변

Thank You !

수고하셨습니다.