Object-Oriented Programming II

객체지향프로그래밍 II

Lecture 7

14장 템플릿 (Templates)

1. Introduction
2. Function Templates
3. Class Templates
4. Nontype Parameters

박인규 교수

1. Introduction

함수 템플릿 (function templates) 및 클래스 템플릿 (class templates)

- ✓ 모든 종류의 자료형을 처리할 수 있는 공통 함수나 클래스를 간단히 설계할 수 있도록 한다.
- ✓ 서로 다른 자료형에 대해 수행되는 여러 종류의 함수나 클래스를 만들어 둘 필요가 없음
- ✓ 특정한 자료형이 아닌, 일반적인 자료형에 대한 프로그래밍 (generic programming)
- ✓ Template은 판화에, template 사용은 판화 찍기에 비유한다.
  ➡ 서로 다른 판화 찍기는 형태는 같지만 색상을 다르게 할 수 있음

2. Function Templates

함수 템플릿 (function templates)

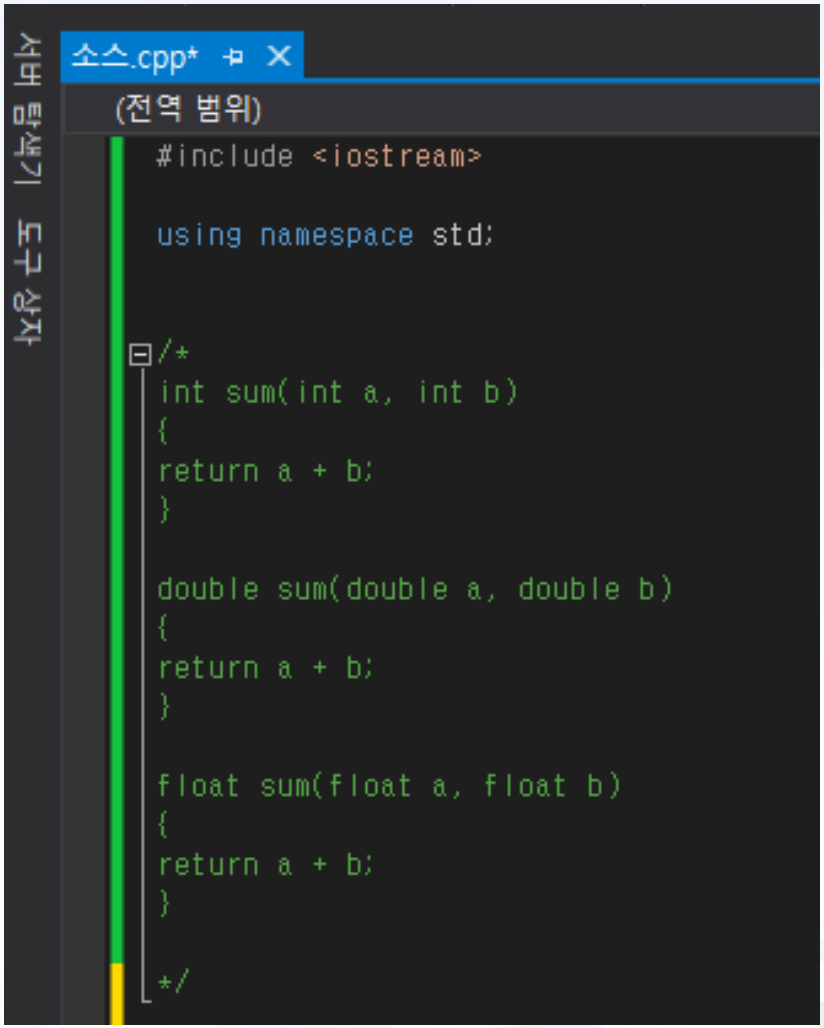
- ✓ 서로 다른 자료형에 대해 같은 일을 수행하는 통일된 오버로드된 함수를 생성하기 위해 사용
- ✓ 즉, 자료형 자체가 함수의 매개 변수임

  - 프로그래머는 template를 이용하여 단 하나의 함수를 작성
    - Template = 틀
  - 컴파일러는 함수 호출시 사용된 인자의 자료형에 대한 함수, 즉 특수화 (specializations)된 함수로 바꾸어 컴파일함
  - 다음 예제 참조

Function-template 정의

- Template 헤더를 사용
  - ✓ template 키워드
  - ✓ Template의 매개변수들
    - Template 매개변수들은 키워드 class 또는 typename 을 이용하여 지정 (순서는 상관없음)하고 < 와 > 를 이용하여 묶어줌
    - 인자의 자료형, 함수의 지역 변수, 반환형을 지정하기 위해 사용됨
  - ✓ 사례
    - template< typename T >
    - template< class ElementType >
    - template< typename BorderType, typename Filltype >

Templates 예제 (참고자료)



Templates 예제 (참고자료)

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

int main()
{

    int result1 = sum(1, 2);

    double result2 = sum(1.0, 2.5);

    float result3 = sum(1.5, 2.5);

    cout << result1 << endl;
    cout << result2 << endl;
    cout << result3 << endl;

    return 0;
}
```

<실행결과>

```
3
3.5
4
계속하려면 아무 키나 누르십시오 . . .
```

Function Template 예제

```
1  // Fig. 14.1: fig14_01.cpp
2  // Using template functions.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // function template printArray definition
8  template< typename T >
9  void printArray( const T *array, int count )
10 {
11    for ( int i = 0; i < count; i++ )
12       cout << array[ i ] << " ";
13
14    cout << endl;
15 } // end function template printArray
16
17 int main()
18 {
19    const int ACOUNT = 5; // size of array a
20    const int BCOUNT = 7; // size of array b
21    const int CCOUNT = 6; // size of array c
22
23    int a[ ACOUNT ] = { 1, 2, 3, 4, 5 };
24    double b[ BCOUNT ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
25    char c[ CCOUNT ] = "HELLO"; // 6th position for null
26
27    cout << "Array a contains:" << endl;
```

Type template parameter **T** specified in template header

Function Template 예제

```
28
29    // call integer function-template specialization
30    printArray( a, ACOUNT );
31
32    cout << "Array b contains:" << endl;
33
34    // call double function-template specialization
35    printArray( b, BCOUNT );
36
37    cout << "Array c contains:" << endl;
38
39    // call character function-template specialization
40    printArray( c, CCOUNT );
41    return 0;
42 } // end main
```

Creates a function-template specialization of **printArray** where **int** replaces **T**

Creates a function-template specialization of **printArray** where **double** replaces **T**

Creates a function-template specialization of **printArray** where **char** replaces **T**

```
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```

3. Class Templates

클래스 템플릿 (class templates)

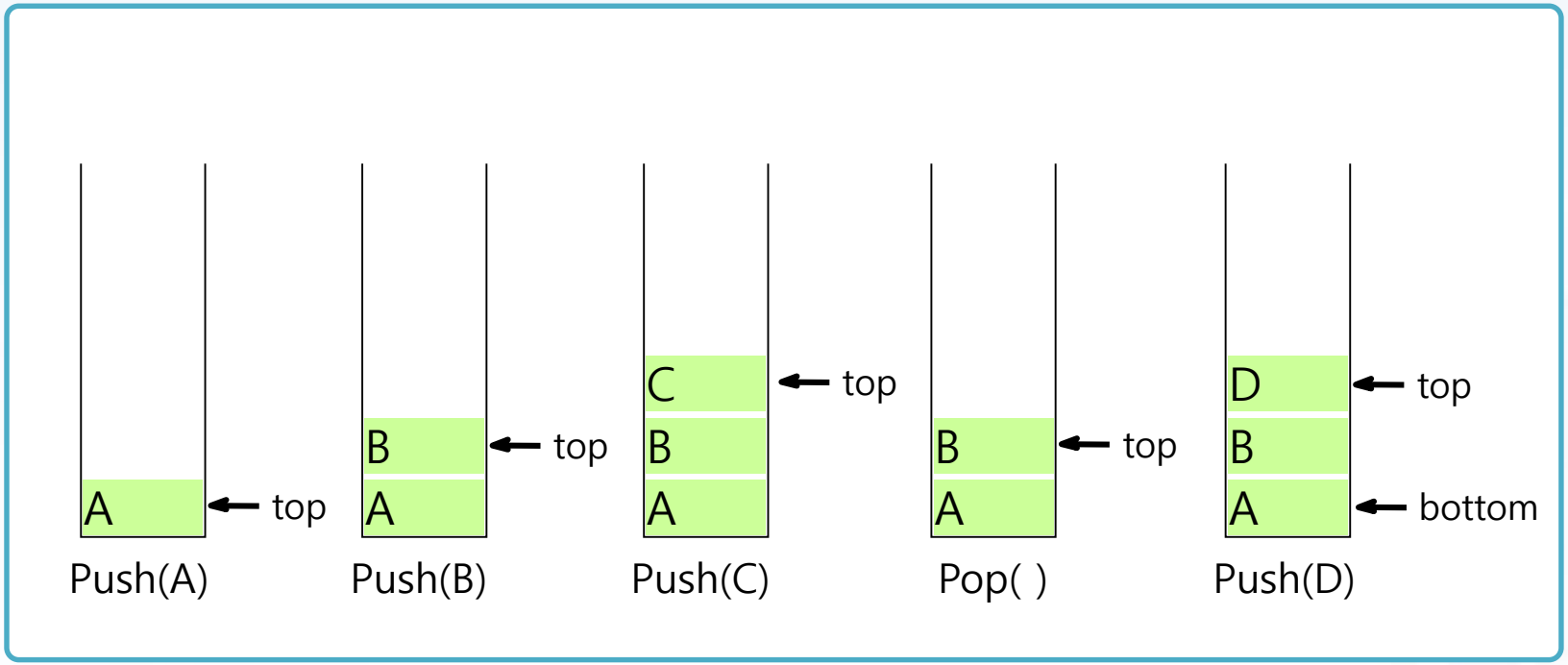
- ✓ 클래스 멤버의 자료형이 매개변수화 됨 (parameterized types)
- ✓ Class-template 정의는 template을 의미하는 헤더가 추가
  - template< typename T > // 다음 예제 참고
  - 이 때, 자료형 매개변수 (type parameter) T 는 멤버 함수와 데이터 멤버의 자료형 지정에 모두 사용할 수 있다.
- ✓ 두 개 이상의 자료형 매개변수를 지정 가능함
  - template< typename T1, typename T2 >

스택(stack) 자료구조

- 순서 리스트(ordered list)의 특수한 한 형태
  - ✓ A = a_0, a_1, …, a_{n-1}
- Top이라고 하는 한쪽 끝에서 모든 삽입(push)과 삭제(pop) 가 일어나는 순서 리스트
  - ✓ 스택 S=(a_0,….,a_{n-1})
    - a_0는 bottom, a_{n-1}은 top의 원소
    - a_i는 원소 a_{i-1}(0<i<n)의 위에 있음
- 後入先出 (LIFO, Last-In-First-Out) 리스트

스택(stack) 자료구조에서 원소의 삽입과 삭제의 동작 과정

- ✓ Push(A)➡Push(B)➡Push(C)➡Pop( )➡Push(D)
- ✓ LIFO, Last-In-First-Out



클래스 Template 예제 (stack.h)

```
1  // Fig. 14.2: Stack.h
2  // Stack class template.
3  #ifndef STACK_H
4  #define STACK_H
5
6  template< typename T >
7  class Stack
8  {
9  public:
10    Stack( int = 10 ); // default constructor (Stack size 10)
11
12    // destructor
13    ~Stack()
14    {
15       delete [] stackPtr; // deallocate internal space for Stack
16    } // end ~Stack destructor
17
18    bool push( const T& ); // push an element onto the Stack
19    bool pop( T& ); // pop an element off the Stack
20
21    // determine whether Stack is empty
22    bool isEmpty() const
23    {
24       return top == -1;
25    } // end function isEmpty
```

Create class template **Stack** with type parameter **T**

Member functions that use type parameter **T** in specifying function parameters

클래스 Template 예제 (stack.h)

```
26
27    // determine whether Stack is full
28    bool isFull() const
29    {
30       return top == size - 1;
31    } // end function isFull
32
33 private:
34    int size; // # of elements in the Stack
35    int top; // location of the top element (-1 means empty)
36    T *stackPtr; // pointer to internal representation of the Stack
37 }; // end class template Stack
38
39 // constructor template
40 template< typename T >
41 Stack< T >::Stack( int s )
42    : size( s > 0 ? s : 10 ), // validate size
43      top( -1 ), // Stack initially empty
44      stackPtr( new T[ size ] ) // allocate memory for elements
45 {
46    // empty body
47 } // end Stack constructor template
```

Data member **stackPtr** is a pointer to a **T**

Member-function template definitions that appear outside the class-template definition begin with the template header

클래스 Template 예제 (stack.h)

```
48
49 // push element onto Stack;
50 // if successful, return true; otherwise, return false
51 template< typename T >
52 bool Stack< T >::push( const T &pushValue )
53 {
54    if ( !isFull() )
55    {
56       stackPtr[ ++top ] = pushValue; // place item on Stack
57       return true; // push successful
58    } // end if
59
60    return false; // push unsuccessful
61 } // end function template push
62
63 // pop element off Stack;
64 // if successful, return true; otherwise, return false
65 template< typename T >
66 bool Stack< T >::pop( T &popValue )
67 {
68    if ( !isEmpty() )
69    {
70       popValue = stackPtr[ top-- ]; // remove item from Stack
71       return true; // pop successful
72    } // end if
73
74    return false; // pop unsuccessful
75 } // end function template pop
76
77 #endif
```

클래스 Template 예제 (driver)

```
1  // Fig. 14.3: fig14_03.cpp
2  // Stack class template test program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Stack.h" // Stack class template definition
8
9  int main()
10 {
11    Stack< double > doubleStack( 5 ); // size 5
12    double doubleValue = 1.1;
13
14    cout << "Pushing elements onto doubleStack\n";
15
16    // push 5 doubles onto doubleStack
17    while ( doubleStack.push( doubleValue ) )
18    {
19       cout << doubleValue << ' ';
20       doubleValue += 1.1;
21    } // end while
22
23    cout << "\nStack is full. Cannot push " << doubleValue
24       << "\n\nPopping elements from doubleStack\n";
25
26    // pop elements from doubleStack
27    while ( doubleStack.pop( doubleValue ) )
28       cout << doubleValue << ' ';
```

Create class-template specialization **`Stack< double >`** where type **`double`** is associated with type parameter **`T`**

클래스 Template 예제 (driver)

```
29
30    cout << "\nStack is empty. Cannot pop\n";
31
32    Stack< int > intStack; // default size 10
33    int intValue = 1;
34    cout << "\nPushing elements onto intStack\n";
35
36    // push 10 integers onto intStack
37    while ( intStack.push( intValue ) )
38    {
39       cout << intValue << ' ';
40       intValue++;
41    } // end while
42
43    cout << "\nStack is full. Cannot push " << intValue
44       << "\n\nPopping elements from intStack\n";
45
46    // pop elements from intStack
47    while ( intStack.pop( intValue ) )
48       cout << intValue << ' ';
49
50    cout << "\nStack is empty. Cannot pop" << endl;
51    return 0;
52 } // end main
```

Create class-template specialization **`Stack< int >`** where type **`int`** is associated with type parameter **`T`**

클래스 Template 예제 (실행 결과)

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

클래스 Template 또다른 예제 (driver)

```
1  // Fig. 14.4: fig14_04.cpp
2  // Stack class template test program. Function main uses a
3  // function template to manipulate objects of type Stack< T >.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include <string>
9  using std::string;
10
11 #include "Stack.h" // Stack class template definition
12
13 // function template to manipulate Stack< T >
14 template< typename T >
15 void testStack(
16    Stack< T > &theStack, // reference to Stack< T >
17    T value, // initial value to push
18    T increment, // increment for subsequent values
19    const string stackName ) // name of the Stack< T > object
20 {
21    cout << "\nPushing elements onto " << stackName << '\n';
22
23    // push element onto Stack
24    while ( theStack.push( value ) )
25    {
26       cout << value << ' ';
27       value += increment;
28    } // end while
```

Use a function template to process **`Stack`** class-template specializations

클래스 Template 또다른 예제 (driver)

```
29
30    cout << "\nStack is full. Cannot push " << value
31       << "\n\nPopping elements from " << stackName << '\n';
32
33    // pop elements from Stack
34    while ( theStack.pop( value ) )
35       cout << value << ' ';
36
37    cout << "\nStack is empty. Cannot pop" << endl;
38 } // end function template testStack
39
40 int main()
41 {
42    Stack< double > doubleStack( 5 ); // size 5
43    Stack< int > intStack; // default size 10
44
45    testStack( doubleStack, 1.1, 1.1, "doubleStack" );
46    testStack( intStack, 1, 1, "intStack" );
47
48    return 0;
49 } // end main
```

클래스 Template 또다른 예제 (실행 결과)

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

4. Nontype Parameters

Nontype template (즉, 고정된 자료형) 매개변수

- ✓ Nontype 의 매개변수를 가질 수 있으며, 이들은 역시 기본 인자(default arguments)를 가질 수 있다.
  - 클래스 생성자를 이용하는 것보다 직관적으로 간단하고 효율적임
- ✓ 상수형 (const)으로 간주됨
- ✓ 예제)
  - Template 헤더

```
template< typename T, int elements >
```

  - Template 선언 (100개짜리 doulble형 데이터를 저장하는 스택 구조)

```
Stack< double, 100 > salesFigures;
```

기본 인자를 가지는 템플릿 매개 변수

✓ 자료형의 매개변수는 역시 기본 인자(즉, 기본 자료형)를 가질 수 있다.

- 예제)

  ➢ Template 헤더

  ```
  template< typename T = string >
  ```

  ➢ Template 선언

  ```
  Stack <> jobDescriptions;
  ```

  : 이와 같이 별도의 자료형 지정 없을 경우 T의 기본값은 string 형 이다.

15주차 1교시 수업을 마칩니다.