Project #2 Servey on Programming Paradigms

Generic Programming

제출일 : 2016.05.23
과목명 : 프로그래밍 언어
소　속 : 숭실대학교 컴퓨터학부
학　번 : 20142418
발표자 : 이주원

CONTENTS

01 개요

02 언어별 제네릭 비교

03 결론

04 참고 문헌

05 Q&A

01 개요

제네릭 프로그래밍

- 분류 : Programming paradigms - Generic
- 데이터 타입을 인자로 사용하여, 하나의 값이 여러 다른 데이터 타입들을 가질 수 있는 기술
- 같은 알고리즘의 비슷한 구현 방식 사이의 공통점을 찾는 것에 초점을 맞춤
- 동일한 알고리즘이나 데이터 구조를 여러 가지 데이터 타입으로 재사용할 수 있으며, 모듈성을 향상시키기 때문에 프로그램 설계에 도움이 됨
- 대부분의 프로그램에 사용됨

02 언어별 제네릭 비교

JAVA Generic Programming

```
public class Entry<KeyType, ValueType> {

    private final KeyType key;
    private final ValueType value;

    public Entry(KeyType key, ValueType value) {
        this.key = key;
        this.value = value;
    }

    public KeyType getKey() {
        return key;
    }

    public ValueType getValue() {
        return value;
    }

    public String toString() {
        return "(" + key + ", " + value + ")";
    }

}
```

Java 제네릭 클래스 생성

- 클래스 이름 옆에 < > 를 사용하여 제네릭 클래스 생성
- < > 안에 선언된 타입들은 클래스 내에서 사용 가능
- 기본 데이터 타입은 전달 불가

02 언어별 제네릭 비교

JAVA Generic Programming

```
Entry<String, String> grade = new Entry<String, String>("Mike", "A");
Entry<String, Integer> mark = new Entry<String, Integer>("Mike", 100);
System.out.println("grade: " + grade);
System.out.println("mark: " + mark);
```

```
public class Entry<KeyType, ValueType>
```

```
grade: (Mike, A)
mark: (Mike, 100)
```

Java 제네릭 클래스 인스턴스 생성

- < > 안에 타입을 전달하여 제네릭 클래스의 인스턴스 생성
- Entry 클래스의 경우, KeyType과 ValueType이 전달된 타입에 각각 대응되어 사용됨

02 언어별 제네릭 비교

JAVA Generic Programming

```
List<String> v = new ArrayList<>();
v.add("test");
Integer i = v.get(0); // (type error)  compilation-time error
```

Compile-time Java 제네릭

- Java SE 7 이후부터 인스턴스 생성 시에 < > 안의 타입들을 생략 가능(Diamond Operator)
- 제네릭 이전 JVM과의 호환을 위해 compile-time에 제네릭 클래스의 타입을 제거(Type erasure, ex. List<String> -> List)
- 따라서 compile-time에 타입을 검사하며 run-time에는 Object 타입으로 실행됨

02 언어별 제네릭 비교

JAVA Generic Programming

```
public class GenericTester
{
    public static void main(String args[])
    {
        Box<Integer> a = new Box<Integer>(10);
        System.out.println(a);
        Box<Integer> b = new Box<Integer>(20);
        System.out.println(b);
        Box<Double> c = new Box<Double>(30.0);
        System.out.println(c);
    }
}

class Box<T>
{
    private T value;
    private static int sValue = 0;

    public Box(T value)
    {
        this.value = value;
        sValue++;
    }

    public String toString()
    {
        String result = "";

        result += "value : " + value + '\n';
        result += "sValue : " + sValue + '\n';

        return result;
    }
}
```

```
value : 10
sValue : 1

value : 20
sValue : 2

value : 30.0
sValue : 3
```

Run-time Java 제네릭

- 제네릭 클래스 마다 단 하나의 복사본만 생성
- 따라서 모든 제네릭 클래스의 인스턴스는 해당 클래스의 static 변수를 공유

02 언어별 제네릭 비교

JAVA Generic Programming

```
class SuperClass
{
    public String toString()
    {
        return "Super Class";
    }
}

class SubClass extends SuperClass
{
    public String toString()
    {
        return "Sub Class";
    }
}
```

```
public static void printAll(ArrayList<SuperClass> al)
{
    for(int i=0; i<al.size(); i++)
        System.out.println(al.get(i));
}
```

```
ArrayList<SuperClass> superClass = new ArrayList<>();
superClass.add(new SuperClass());

ArrayList<SubClass> subClass = new ArrayList<>();
superClass.add(new SubClass());

printAll(superClass);
printAll(subClass); // Error!
```

Type wildcard

- 함수의 인자가 제네릭 클래스의 인스턴스일 때, 해당 타입 인자와 상속 관계에 있는 타입을 인자로 하는 제네릭 클래스의 인스턴스는 전달될 수 없음

02 언어별 제네릭 비교

JAVA Generic Programming

```
public static void printAll(ArrayList<? extends SuperClass> al)
{
    for(int i=0; i<al.size(); i++)
        System.out.println(al.get(i));
}
```

```
printAll(superClass);
printAll(subClass); // not error
```

Type wildcard

- <?> : 모든 클래스
- <? extends ClassName> : ClassName 클래스이거나 ClassName을 상속하는 클래스
- <? super ClassName> : ClassName 클래스이거나 ClassName에게 상속되는 클래스

02 언어별 제네릭 비교

JAVA Generic Programming

```
ArrayList<Integer> li = new ArrayList<Integer>();
ArrayList<Float> lf = new ArrayList<Float>();
if (li.getClass() == lf.getClass()) { // evaluates to true
    System.out.println("Equal");
}
```

Java generic programming

- 타입 검사를 compile-time에 하기 때문에 안전하며 run-time에 에러가 발생하지 않음
- Compile-time에 타입이 제거되기 때문에 run-time에 타입에 대한 정보를 얻을 수 없음

02 언어별 제네릭 비교

C++ Template

```
template <class identifier>
template <typename identifier>
```

C++ 템플릿

- template <class > 혹은 template <typename >을 통해 템플릿 생성
- 클래스 뿐만 아니라, 기본 데이터 타입들도 전달될 수 있기 때문에, 일반적으로 덜 혼란스러운 typename을 사용

02 언어별 제네릭 비교

C++ Template

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

```
std::cout << max(3, 7.0) << std::endl;
std::cout << max<double>(3, 7.0) << std::endl;
```

C++ 함수 템플릿

- 함수 정의 앞에 template 키워드를 사용하여 함수 템플릿 정의 가능
- 호출할 때 타입을 전달하지 않으면 컴파일러가 타입을 추론
- 그러나 컴파일러에 따라 문제가 발생할 수 있기 때문에, 타입을 전달하는 편이 더 안전함

02 언어별 제네릭 비교

C++ Template

```
template <typename T>
class Box
{
private:
    T value;
public:
    Box(T v)
    {
        value = v;
    }
    T getValue()
    {
        return value;
    }
};
```

C++ 클래스 템플릿

- 클래스 정의 앞에 template 키워드를 사용하여 클래스 템플릿 정의 가능
- typename으로 선언된 키워드는 클래스 내에서 사용 가능

02 언어별 제네릭 비교

C++ Template

```
template <>
bool max<bool>(bool a, bool b) {
    return a || b;
}
```

Template specialization

- 템플릿을 사용할 때 특정 타입 인자에 따른 특수화가 가능
- 제네릭 함수/클래스에 대해 특수화된 타입을 전달하면 기존의 함수/클래스를 사용하지 않고 따로 생성한 함수/클래스를 사용
- 특정 타입 인자에 대한 코드 최적화 가능

02 언어별 제네릭 비교

C++ Template

```
template <typename Key, typename Value>
class KeyValuePair {};
```

```
template <typename Key>
class KeyValuePair<Key, std::string> {};
```

Partial specialization

- 함수/클래스를 특수화 할 때 둘 이상의 타입 인자에 대해 부분적으로 특수화가 가능
- 위와 같은 경우 두 번째 타입 인자가 std::string인 경우 특수화된 클래스를 사용

02 언어별 제네릭 비교

C++ Template

C++ template

- 사용되는 타입 마다 코드를 생성하기 때문에 실행 파일의 크기가 커짐

- Compile-time에 가능한 한 많은 연산을 처리하기 때문에 컴파일 속도가 느리고 실행 속도가 빠름

02 언어별 제네릭 비교

Python Generic Programming

```
class BinaryTree:
    def __init__(self, left, right):
        self.left, self.right = left, right
```

```
branch1 = BinaryTree(1,2)
myitem = MyClass()
branch2 = BinaryTree(myitem, None)
tree = BinaryTree(branch1, branch2)
```

Python Generic

- 파이썬의 경우, 기본적으로 동적 타이핑 방식이므로, 사실상 제네릭 프로그래밍이라는 개념이 존재하지 않음(기본적으로 포함)
- run-time에 타입을 결정하기 때문에 속도가 느리고 타입 안정성이 떨어짐

02 언어별 제네릭 비교

C Generic Programming

- C Generic 구현 방식

1. void 포인터
   함수 포인터

Size 전달

2. 매크로

Type 전달

02 언어별 제네릭 비교

C Generic Programming

1. void 포인터, 함수 포인터

```
typedef struct _listNode {
  void *data;
  struct _listNode *next;
} listNode;


typedef struct {
  int logicalLength;
  int elementSize;
  listNode *head;
  listNode *tail;
  freeFunction freeFn;
} list;
```

Linked List 데이터 구조 정의

- 노드의 데이터를 void 포인터로 저장
- elementSize : 저장할 데이터의 크기
- freeFn : 저장할 데이터에 대한 free 함수

02 언어별 제네릭 비교

C Generic Programming

1. void 포인터, 함수 포인터

```
void list_new(list *list, int elementSize, freeFunction freeFn)
{
  assert(elementSize > 0);
  list->logicalLength = 0;
  list->elementSize = elementSize;
  list->head = list->tail = NULL;
  list->freeFn = freeFn;
}
```

Linked List 생성 함수

- elementSize : 데이터 크기를 전달
- freeFn : free 함수 전달

02 언어별 제네릭 비교

C Generic Programming

1. void 포인터, 함수 포인터

```
void list_append(list *list, void *element)
{
  listNode *node = malloc(sizeof(listNode));
  node->data = malloc(list->elementSize);
  node->next = NULL;

  memcpy(node->data, element, list->elementSize);

  if(list->logicalLength == 0) {
    list->head = list->tail = node;
  } else {
    list->tail->next = node;
    list->tail = node;
  }

  list->logicalLength++;
}
```

Linked List 추가 함수

- 새로 생성한 노드의 데이터 변수에 elementSize 크기로 메모리 할당
- 인자로 들어온 element를 elementSize만큼 노드의 데이터에 메모리 복사

-> 다른 함수들도 비슷한 방식으로 구현 가능

02 언어별 제네릭 비교

C Generic Programming

1. void 포인터, 함수 포인터

```
int numbers = 10;
printf("Generating list with the first %d positive numbers...\n", numbers);

int i;
list list;
list_new(&list, sizeof(int), NULL);

for(i = 1; i <= numbers; i++) {
  list_append(&list, &i);
}
```

Linked List 사용 예제(int)

- 리스트 생성 함수에 int의 크기와 NULL 전달(데이터가 int형이므로 free 함수 필요 없음)
- 리스트 추가 함수의 인자가 void 포인터이므로 변수의 주소값 전달

02 언어별 제네릭 비교

C Generic Programming

1. void 포인터, 함수 포인터

```
void swap(void *p, int obj_size, int i, int j) {
  void *tmp = malloc(obj_size); // for efficiency, we can preallocate
  memcpy(tmp, p+obj_size*i, obj_size);
  memcpy(p+obj_size*i, p+obj_size*j, obj_size);
  memcpy(p+obj_size*j, tmp, obj_size);
  free(tmp);
}
```

void 포인터, 함수 포인터를 사용한 구현 방식

- 매크로를 사용한 구현 방식에 비해 가독성이 좋음
- 값을 대입할 때마다 메모리 카피를 해야 하기 때문에, 저장할 데이터의 크기가 작을 경우 비효율적임
- 데이터를 저장할 때 void 포인터 변수가 추가적으로 필요하기 때문에 메모리 관리에도 비효율적임

02 언어별 제네릭 비교

C Generic Programming

2. 매크로

```
#define STACK_DECLARE(type)
typedef struct stack_##type##_s {
type data;
struct stack_##type##_s *next;
} stack_##type ;
```

Linked List Stack 데이터 구조 정의

- 매크로 사용 시 데이터의 타입을 전달
- 전달 받은 타입으로 데이터 선언

02 언어별 제네릭 비교

C Generic Programming

2. 매크로

```
typedef struct stack_int_s {
int data;
struct stack_int_s *next;
} stack_int;
```

STACK_DECLARE(int)

```
typedef struct stack_double_s {
double data;
struct stack_double_s *next;
} stack_double;
```

STACK_DECLARE(double)

- 전달 받은 데이터 타입으로 구조체가 정의됨

02 언어별 제네릭 비교

C Generic Programming

2. 매크로

```
#define STACK_DEFINE(type)
void stack_##type##_push(stack_##type **stack, type data) {
stack_##type * new_node = malloc(sizeof(*new_node));
if (NULL == new_node) {
fputs("Couldn't allocate memoryn", stderr);
abort();
}
new_node->data = data;
new_node->next = *stack;
*stack = new_node;
}

type stack_##type##_pop(stack_##type **stack){
if (NULL == stack || NULL == *stack){
fputs("Stack underflow.n", stderr);
abort();
}
stack_##type *top = *stack;
type value = top->data;
*stack = top->next;
free(top);
return value;
}
```

Linked List Stack 함수 정의

- 전달 받은 데이터 타입에 따른 push, pop 함수를 정의

02 언어별 제네릭 비교

C Generic Programming

2. 매크로

```
/* Expansion if int is supplied as the macro argument */
void stack_int_push(stack_int **stack, int data) {
stack_int * new_node = malloc(sizeof (*new_node));
if (NULL == new_node) {
fputs("Couldn't allocate memoryn", stderr);
abort();
}
new_node->data = data;
new_node->next = *stack;
*stack = new_node;
}int stack_int_pop(stack_int **stack) {
if (NULL == stack || NULL == *stack) {
fputs("Stack underflow.n", stderr);
abort();
}
stack_int *top = *stack;
int value = top->data;
*stack = top->next;
free(top);
return value;
}
```

STACK_DEFINE(int)

- int형에 대한 push, pop 함수의 정의를 생성

02 언어별 제네릭 비교

C Generic Programming

2. 매크로

```
#define STACK_ALLOCATE(type)
(stack_##type*)malloc(sizeof(stack_##type))

#define STACK_TYPE(type)
stack_##type

#define STACK_DATA(stack)
(stack)->data

#define STACK_PUSH(type, stack, data)
stack_##type##_push(stack, data)

#define STACK_POP(type, stack)
stack_##type##_pop(stack)
```

Linked List Stack 함수 wrapper

- 기본적인 연산들과 정의한 함수들을 매크로로 선언 (사용의 일관성)

02 언어별 제네릭 비교

C Generic Programming

2. 매크로

```
STACK_DECLARE(int)
STACK_DEFINE(int)
STACK_DECLARE(double)
STACK_DEFINE(double)

int main(int argc, char** argv)
{
    int i;

    /* New stack . Alaways assign NULL */
    STACK_TYPE(int) * st = STACK_ALLOCATE(int);
    STACK_TYPE(double) * st2 = STACK_ALLOCATE(double);

    for (i = 0; i < 100; ++i) {
        printf("PUSH: %d\n", i);
        STACK_PUSH(int, st, i);
        STACK_PUSH(double, st2, i);
    }

    while (i-- > 0) {
        printf("POP: %d %2.2f\n", STACK_POP(int, st),
            STACK_POP(double, st2));
    }
```

Linked List Stack 사용 예제

- int형, double형에 대한 스택 구조체, 함수 정의

- int형, double형 스택 선언 및 메모리 할당, Push, Pop

```
#define STACK_ALLOCATE(type)
(stack_##type*)malloc(sizeof(stack_##type))

#define STACK_TYPE(type)
stack_##type

#define STACK_DATA(stack)
(stack)->data

#define STACK_PUSH(type, stack, data)
stack_##type##_push(stack, data)

#define STACK_POP(type, stack)
stack_##type##_pop(stack)
```

02 언어별 제네릭 비교

C Generic Programming

2. 매크로

Program	Version	IMIC (sec)	IMIM (MB)	IMSC (sec)	IMSC (MB)	ILIC (sec)	ILIM (MB)	ILSC (sec)	ILSM (MB)	Comments
libavl_rb_cpp	2.0.2	6.66	19.87	14.79	19.87	8.36	29.41	17.40	29.41	C++, rb
libavl_rb	2.0.2	10.77	19.87	20.23	19.87	10.70	48.41	20.45	48.41	C void*, rb
NP_rbtree	1.7	6.24	19.87	13.76	19.87	8.36	29.41	16.15	39.19	C macro, rb

매크로를 사용한 구현 방식

- void 포인터 구현 방식에 비해 가독성이 좋지 않음
- void 포인터 구현 방식보다 속도가 빠르며 C++ 템플릿을 이용한 구현과 성능이 비슷함

03 결론

제네릭 프로그래밍

- 제네릭 프로그래밍도 언어마다 구현 방식과 특징이 다름
- Java : Type erasure, Type wildcard, 하나의 복사본
- C++ : 함수/클래스 특수화, 느린 컴파일 빠른 실행, 여러 개의 복사본
- Python : 동적 타이핑, 기본적으로 제네릭을 포함, 낮은 타입 안정성
- C : void 포인터, 함수 포인터 – 높은 가독성, 낮은 성능
  매크로 – 낮은 가독성, 좋은 성능

04 참고 문헌

- 개요
https://en.wikipedia.org/wiki/Generic_programming
- 개요
https://ko.wikipedia.org/wiki/%EC%A0%9C%EB%84%A4%EB%A6%AD_%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D
- Java 제네릭
https://en.wikipedia.org/wiki/Generics_in_Java
- C++ 템플릿
https://en.wikipedia.org/wiki/Template_(C%2B%2B)
- Python 제네릭
http://stackoverflow.com/questions/6725868/generics-templates-in-python
- C 제네릭 void 포인터, 함수 포인터 구현
http://pseudomuto.com/development/2013/05/02/implementing-a-generic-linked-list-in-c/
- C 제네릭 매크로 구현
http://andreinc.net/2010/09/30/generic-data-structures-in-c/
- C 제네릭 평가
https://attractivechaos.wordpress.com/2008/10/02/using-void-in-generic-c-programming-may-be-inefficient/

Q & A

Generic Programming