PUSAN NATIONAL UNIVERSITY

REPORT

프로그래밍언어론 과제4

과 목 명 :	프로그래밍언어론
지도교수 :	
학 과 :	전기컴퓨터공학부 정보컴퓨터공학전공
학 번 :	
이 름 :	장 수현
제 출 일 :	2020년 5월 1일

부산대학교

1. For an elementary data type in a language with which you are familiar, do the following:

1.1 Describe the set of values that data objects of that type may contain.

1.2 Determine the storage representation for values of that type (used in your local implementation of the language).

1.3 Define the syntactic representation used for constants of that type.

1.4 Determine the set of operations defined for data objects of that type; for each operation, give its signature and syntactic representation on the language.

1.5 For each operation, determine whether it is implemented through software simulation or directly as a single-hardware instruction.

1.6 Describe any attributes that a data object of that type may have other than its data type.

1.7 Determine if any of the operation symbols or names used to represent operations of the type are overloaded. For each overloaded operation name, determine when (compile time or run time) the specific meaning of each use of the overloaded name in a statement is determined.

1.8 Determine whether static or dynamic type checking is used to determine the validity of each use of each operation of the type.

sol)

C언어를 기준으로 풀이 작성.

1.1

(1) Scalar Types

정수

- long long : $-2^{63} \sim 2^{63}-1$
- unsigned long long : $0 \sim 2^{64}-1$
- long : $-2^{31} \sim 2^{31}-1$
- unsigned long : $0 \sim 2^{32}-1$
- int : $-2^{31} \sim 2^{31}-1$
- unsigned int : $0 \sim 2^{32}-1$
- short : $-2^{15} \sim 2^{15}-1$
- unsigned short : $0 \sim 2^{16}-1$
- char : $-2^{7} \sim 2^{7}-1$
- unsigned char : $0 \sim 2^{8}-1$

실수

- double : $-1.7*10^{308} \sim 1.7*10^{308}$
- float : $-3.4*10^{38} \sim 3.4*10^{38}$

포인터

- pointer : $0 \sim 2^{word\_size}-1$

(2) Composite Types

배열 : 가변적. 요소가 기준.

구조체 : 가변적. 요소가 기준.

1.2

(1) Scalar Types

정수

- long long : 이진수, 2의 보수, 8Bytes
- unsigned long long : 이진수, 2의 보수, 8Bytes
- long : 이진수, 2의 보수, 4Bytes
- unsigned long : 이진수, 2의 보수, 4Bytes
- int : 이진수, 2의 보수, 4Bytes
- unsigned int : 이진수, 2의 보수, 4Bytes
- short : 이진수, 2의 보수, 2Bytes
- unsigned short : 이진수, 2의 보수, 2Bytes
- char : 이진수, 2의 보수, 1Bytes
- unsigned char : 이진수, 2의 보수, 1Bytes

실수

- double : 이진수, 부동 소수(IEEE754 double precision), 8Bytes
- float : 이진수, 부동 소수(IEEE754 single precision), 4Bytes

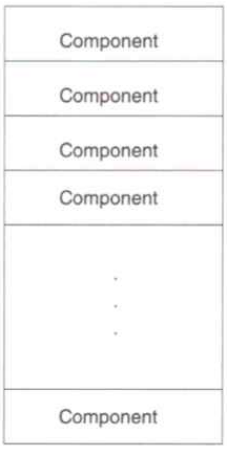
포인터

- pointer : 16진수, 2의 보수, word_size/8Bytes

(2) Composite Types

배열 : 그림1처럼 같은 타입의 요소가 연속된 형태로 저장.

구조체 : 그림1처럼 다른 타입의 요소가 연속된 형태로 저장. (패딩 이용)



SEQUENTIAL

그림 1 연속된 저장 형태

1.3

C언어에서 상수를 표현하는 방법에는 다음의 세 가지가 있다.

(1) literal

그 자체로 상수인 것.

단, 유지보수의 효율성을 위해 사용하지 않는 것이 권장된다.

ex) 5(정수 상수), 7.17(실수 상수), "Programming Language"(문자열 상수)

(2) #define 매크로

특정 값(상수)에 이름을 붙여 매크로 정의.

단, 이때 매크로의 이름은 대문자와 _로만 구성되어야 한다.

ex) #define PI 3.14

(3) const 키워드

바뀌지 않을 값을 가지는 변수에 대해 const 키워드를 붙여 상수화.

단, 상수 변수에 대해 선언과 동시에 초기화를 해주어야 한다.

ex) const int Max = 100;

1.4

(1) Scalar Types

: 산술 연산, 관계 연산, 논리 연산, 대입 연산, sizeof 연산, 타입캐스팅, 포인터 등이 지원된다.

(2) Composite Types

: Scalar Types에 비해 적은 연산이 지원된다. 배열이나 구조체만을 위한 연산자도 있다.

*배열 - 요소 참조 연산(array[i]), sizeof 연산 등

*구조체 - 요소 참조 연산(struct.a), 포인터 참조 연산(p_struct->struct.a),
대입 연산(C99부터 적용) 등

1.5

(1) 하드웨어로 구현된 연산

: 산술 연산, 관계 연산, 논리 연산, 대입 연산(구조체의 경우는 소프트웨어로 구현) 등

(2) 소프트웨어로 구현된 연산

: 포인터 증감 연산(포인터 증감할 때 내부적으로 참조 타입의 크기가 곱해진다.),
배열 요소 참조(증감) 연산(배열 인덱싱할 때 내부적으로 요소 타입의 크기가 곱해진다.) 등

1.6

(1) 같은 타입 내에 구분하는 관점

: 같은 타입끼리 비교했을 때 구분되는 해당 타입만의 고유한 속성으로 타입이 가지는 메모리 주소, 참조 타입, 요소 타입, 타입의 크기 등이 있다.

ex)

int a = 3; vs int b = 3; // a와 b는 모두 3이라는 int값을 가지지만 메모리의 주소가 다르다.

int* pa; vs double* pb; // pa와 pb는 참조 타입이 다르다.

int x[3]; vs int y[5]; // x와 y는 길이와 타입의 크기가 다르다.(길이:3,5 & 타입크기:12,20)

(2) 다른 타입 내에 구분하는 관점.

: 1.2에 따라 서로 다른 타입끼리 비교하여 구분할 수 있다.

1.7

산술 연산 중 뺄셈 연산자를 예로 들 수 있다. 서로 다른 타입에 대해 뺄셈 연산을 호출하게 되면, 해당 타입에 맞는 연산이 수행된다.

ex) 3 - 5;를 수행할 경우 정수 뺄셈 연산이 수행된다.

3.0 - 5.0;을 수행할 경우 실수 뺄셈 연산이 수행된다.

오버로딩된 뺄셈 연산자가 확정이 되는 시기가 컴파일 타임인지, 런 타임인지 알아보기 위해 다음과 같이 서로 다른 타입에 대한 뺄셈을 수행하는, 에러가 있는 프로그램을 작성했다. 실행시키지 않고, 컴파일만 했는데 에러가 발생한 것으로 보아 컴파일 타임에 타입을 확인하고 해당하는 타입에 대한 연산을 선택을 한다.

그림 2 오버로딩된 뺄셈 연산자가 언제 확정되는지 확인하기 위한 프로그램 실행 결과 화면.

1.8

C언어는 문법이 제대로 기술되었는지 확인하기 위해 대부분의 static type checking을 지원한다. 그러나 일부는 static type checking으로 확인할 수 없고, defect를 검출해내지 못한다. 그래서 심각한 문제를 발생시킬 수도 있다. 예로 union, void*, type casting 연산 등은 타입검사 기능을 무력화시킬 수 있다. 그러므로 C언어는 강형언어(strongly-typed language)가 아니다.

2. For an elementary data type in a language with which you are familiar, do the following:

2.1 Explain the difference among the type, variables of that type, and constants of that type.

2.2 Show a situation during execution where a data object of that type exists that is neither a variable nor a constant.

2.3 Explain the difference between data objects of that type and the values that those data objects may contain.

sol)

2.1

(1) 변수

- 데이터 객체로, 값을 가지는 메모리 공간을 의미한다.
- 초기화된 이후라도 값의 변경이 가능하다.
- 기본 속성으로 타입, 주소값(R-value), 값(L-value), 이름을 가진다.
- 추가 속성으로 영역, 수명 등을 가진다.

(2) 상수

- 변수의 일종으로, 일반적으로 값만을 의미한다.
- 선언과 동시에 반드시 초기화되어야 한다. (value binding)
- 값이 초기화된 이후에는 변경이 불가능하다.

2.2

프로그램이 실행 중일 때, 변수도 상수도 아닌 데이터 객체는 temporary object의 형태로 존재할 수 있다. 상황을 예로 들면 함수 호출 후 반환을 할 때, 상수를 바로 반환하거나, 변수를 생성해서 값을 담아 전달하는 것이 아니라 순간적으로 유효한 임시변수를 생성해 값을 담아 전달하는 프로그램을 생각하면 된다.

2.3

서로 다른 두 데이터 객체는 타입이든, 이름이든 두 객체 사이에는 어떠한 사소한 차이라도 존재해야 한다. 즉, 같은 데이터 객체가 중복으로 존재할 수 없다는 것이다. 그러나 데이터 객체가 가지는 값은 완전히 동일하여 중복으로 존재할 수 있다.

3. For a language with which you are familiar and uses static type checking, give two examples of constructs that cannot be checked statically. For each construct, determine by running a test program whether the language implementation provides dynamic type checking or leave the construct unchecked during execution.

You use C as a language.

sol)

3.1 statically check를 하지 않는 두 가지 예

C언어는 기본적으로 static type checking을 지원하지만 강형언어(strongly-typed language)가 아니다. 즉, 항상 모든 타입에 대해 statically check를 하는 것이 아니다. C언어를 강형언어라고 할 수 없게 하는 대표적인 예로 union, void*, type casting 연산이 있다.

union의 경우, 한 객체 안에 내부적으로 서로 다른 타입의 객체들을 가지는데, 해당 객체들은 같은 메모리 공간을 공유하며 필요에 따라 하나가 선택되어 쓰인다. 그래서 객체들이 공유하는 메모리 공간에 들어있는 값이 어떻게 해석되는지는 타입에 따라 달라진다. 이러한 특징을 가지는 union은 정적 타입 검사를 하지 않는데, 어떤 타입의 해석방법으로도 값이 해석될 여지가 있기 때문에 프로그래머가 명확히 지정해주지 않으면 문제가 된다.

또한 void*과 type casting를 사용했을 때 오류가 있어도 정적검사를 하지 않아 문제가 된다. void 포인터는 다른 포인터 타입간의 자유로운 변환을 위해 사용이 되는데, 나중에 변환될 것을 고려해 그냥 두었다가 정적검사를 받지 못해 문제를 발생시키게 되는 경우가 있다. 따라서 가능하면 void는 포인터는 사용하지 않고 처음부터 명확하게 타입을 명시해주는 것이 바람직하다.

3.2 statically check를 하지 않는 것의 dynamic check 여부

C언어는 강형 언어가 아니므로, 항상 오류를 탐지할 수 없다. 만일 3.1에서 말한 예들이 모두 정적으로는 타입 검사되진 않지만 동적으로라도 체크를 한다면 타입에 대한 오류를 항상 탐지해낼 수 있을 것이다. 그런데 C언어는 그렇지 못해 때때로 프로그램이 위험성을 가지게 한다. 따라서 C언어로 만든 프로그램은 일부 dynamically checking도 하지 않아 결과적으로 어떠한 타입 검사도 하지 않는 경우가 발생한다.

4. For a language that provides a pointer type for programmer-constructed data objects and operations such as new and dispose, which allocate and free storage for data objects, write a program segment that generates garbage (in the storage management sense). Write a program that generates a dangling reference. If one or the other program segment cannot be written, explain why.

You can find an example in C programming language

sol)

4.1 C언어로 작성한 프로그램이 Garbage와 Dangling reference를 발생시킬 수 있는 이유

Garbage는 데이터 객체에 대한 모든 접근 경로가 없어진 후에도 데이터 객체가 메모리에 계속 남아 있는 것이다. garbage는 메모리를 차지하고 있지만, 접근 경로가 없으므로 사용이 불가능하므로 메모리 관리에 문제가 된다.

Dangling Reference는 포인터 변수가 더이상 의미 없는 데이터를 가리키는 것이다. 데이터를 가리키는 것 자체는 문제가 되지 않지만, 의미가 없는 데이터가 다른 프로그램 혹은 객체에서 중요한 것이 되었을 때 참조하게 되면 치명적인 문제가 된다.

C언어에는 포인터라는 메모리 주소를 직접 건드릴 수 있는 강력한 기능이 있다. 섬세하게 하드웨어를 관리할 수 있다는 점에서 매우 유용하지만, garbage와 dangling reference를 발생시킬 수 있다는 위험도 있다. 따라서 C언어는 포인터가 없는 일부 다른 언어와는 다르게 이 두 가지 문제를 야기시킬 수 있기 때문에 포인터 사용에 있어서 각별히 주의해야 한다.

4.2 Garbage를 발생시키는 프로그램

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    {
        int *p = (int *)malloc(sizeof(int));
        *p = 12345;
    }

    return 0;
}
```

위의 프로그램은 Garbage를 발생시키는 프로그램이다. 포인터 p에 malloc으로 할당된 메모리의 주소를 저장했다. 이후 해당 주소에 12345라는 정수로 값을 초기화하여 유의미한 공간으로 만들었다. 그러나 블록이 끝남과 동시에 p가 할당 해제된다. 12345라는 값을 가진 메모리 공간에 참조할 수 있었던 유일한 포인터인 p가 해제되면서 해당 공간을 참조하는 것이 불가능해진다. 따라서 12345가 저장된 메모리 공간은 Garbage가 되고 메모리 leak을 발생시키게 된다. 그러므로 메모리 공간 활용의 효율성을 위해 Garbage를 생성하지 않도록 주의해야 한다.

4.3 Dangling Reference를 발생시키는 프로그램

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int *pa;
   {
      int *pb = (int *)malloc(sizeof(int));
      pa = pb;
      *pb = 12345;
      free(pb);
   }

   *pa+=1;

    return 0;
}
```

위의 프로그램은 Dangling Reference를 발생시키는 프로그램이다. 포인터 pb는 int만큼의 메모리 공간을 할당하여 그 주소값을 저장하고, 그런 pb의 값을 pa가 저장함으로써 pa도 해당 메모리 공간을 가리키게 된다. 그리고 pb를 해제하게 되면, pa는 pb의 값이었던 12345의 주소값을 저장은 하지만 가리키는 대상이 사라지게 된다. 즉, 더이상 의미가 없어진 12345라는 데이터를 여전히 가리키게 되는 것이다. 그래서 pa는 dangling을 참조하는 dangling pointer가 된다. 만약 이후, 해당하는 주소값에 건드리면 안되는 프로그램이나 값이 저장되고, pa가 이것을 참조하게 된다면 아주 치명적인 오류를 발생시키게 된다. 따라서 프로그램의 안전성을 위해 dangling reference를 발생시키지 않도록 주의해야 한다.