객체지향프로그래밍 숙제 #4 해답

강대기

2007년 12월 13일

제 1 절 숙제

1.1 연습 문제 숙제

1.1.1 포인터 및 함수 선언 (문제 2.3.1)

다음의 선언이나 정의, 또는 명령문들이 의미하는 바를 설명하라.

1. C 스타일 스트링에 대한 포인터 10 개의 배열
2. 10 개의 문자들의 배열에 대한 포인터
3. C 스타일 스트링에 대한 포인터 10 개의 배열
4. C 스타일 스트링에 대한 포인터 10 개의 배열의 포인터
5. `double` 값 2 개의 배열에 대한 포인터 5개의 배열
6. `double` 값의 5x2 이차원 배열에 대한 포인터, 더 정확히 말하자면 `double` 값 2 개로 이루어진 배열이 5 개가 있는 배열에 대한 포인터
7. `"Busan"` 이라는 스트링 리터럴에 대한 포인터
8. `"Busan"` 이라는 상수 스트링 리터럴에 대한 포인터
9. `"Busan"` 이라는 스트링 리터럴에 대한 포인터 상수
10. `"Busan"` 이라는 상수 스트링 리터럴에 대한 포인터 상수
11. `g` 는 `float` 값에 대한 포인터를 반환하는 함수이며, `h` 는 `float` 값을 반환하는 함수 포인터
12. `ex` 는 아무 것도 반환하지 않는 함수로, `int` 값과 바로 뒤에서 설명할 특정 함수 포인터를 전달 인자로 가짐. 그 특정 함수 포인터는 `int` 값을 전달 인자로 받고 `double`을 반환함.
13. `p` 는 함수 포인터이다. 그 함수는 아무 것도 반환하지 않으며 3 개의 전달 인자를 가지는 데, 처음 2 개는 `float` 값이다. 세 번째 전달 인자는 함수 포인터로, 그 함수는 두 개의 `float` 값을 전달 인자로 가지며 `float` 값을 반환한다.

14. p는 포인터로 함수를 가리킨다. 즉 함수 포인터이다. p가 가리키는 함수는 포인터를 반환한다. 그 포인터는 int 를 반환하는 함수의 포인터이다. 즉, p는 int를 반환하는 함수의 포인터를 반환하는 함수의 포인터이다.

15. func는 포인터 5 개를 가진 배열이다. 그 포인터는 int를 반환하는 함수의 포인터이다.

16. p는 포인터 상수이다. p가 가리키는 것은 포인터 상수인데, 그 포인터 상수가 가리키는 것은 문자이다. 즉, 문자 또는 문자열에 대한 포인터 상수를 가리키는 포인터 상수이다.

17. 바로 위의 경우에 더해서, p는 문자 상수 또는 문자열 상수에 대한 포인터 상수를 가리키는 포인터 상수이다

18. p는 형이 정해지지 않은 포인터이다. 이 명령문은 그 p에 double 값 3 개로 이루어진 배열에 대한 포인터를 할당한 것이다.

19. atexit는 함수 포인터를 전달 인자로 받아서 정수를 반환하는 함수이다. 그 함수 포인터는 아무 것도 전달 인자로 받지 않고, 아무 것도 반환하지 않는다.

20. signal은 함수로 int와 함수 포인터를 전달인자로 받는다. 이 함수 포인터는 int를 전달 인자로 받고 아무 것도 반환하지 않는다. signal 함수는 함수 포인터를 반환하는 데, 이 함수 포인터는 int를 전달 인자로 받고 아무 것도 반환하지 않는다.

다음의 설명에 따라 함수 원형을 작성하라.

1. `void func1(void)`
2. `int func2(double)`
3. `char * func3(int &);`
4. `Arff & func4(const Arff &, const Map &, int);`
5. `int func5(const char *);`
6. `void func6(Arff & (*)(Arff &, Map &, int),double);`
7. `int (*func7(Arff & (*)(Arff &, Map &, int), void (*)(Arff & (*)(Arff &, Map &, int),double))) (char *);`

1.1.2 코드 읽기 (문제 2.3.2)

1. 이 프로그램 코드는 '앤드류 쾨니히'의 'C 함정과 실수'라는 책에서 나오는 다음의 예이다.

   해석을 해보자면, 일단 `(*(void(*)())0)();`에서 *, (), ; 또는 void같은 예약어가 아닌 것은 0 뿐이다. 그러므로 우리는 0부터 해석을 시작해야 할 것이다. 우선 0는 그 앞에 `(void(*)())`이 있다. 이렇게 숫자 앞에 ()이 오는 경우는 다음과 같은 타이프 캐스팅(type casting) 즉 형 강제의 경우이다.

```
int x = (int) 3.2;
```

즉 0이 특정 형으로 변환되는 것이다. 그 변환되는 형은 0 앞에 있는 (void(*)())의 괄호 안의 void(*)() 이다. void(*)()는 아무것도 반환하지 않는 함수의 포인터 형임을 알 수 있다. 따라서 (void(*)())0 은 0을 아무것도 반환하지 않는 함수의 포인터로 타이프 캐스팅한 것이다. 이런 경우 0이 함수 포인터가 된 것이므로, 0 번지에 들어있는 주소는 그 함수의 시작 번지가 되는 것이다.

이제 (*(void(*)())0)(); 에서 (void(*)())0 를 fp 로 대치해 보자.

```
(*fp)(); // fp 는 (void(*)())0
```

이것은 함수 포인터 fp 가 가리키는 함수를 호출하는 것이다.

결과적으로 (*(void(*)())0)(); 는 0 번지에 들어있는 주소를 함수의 시작 번지로 삼아서 호출하는 (원래의 의도는 점프하는) 명령문이다.

2. 이 프로그램은 자기 소스 코드 자체를 그대로 다시 출력한다. 이러한 프로그램을 자기 재생산 코드라고 하며, 과거 자기 참조를 연구했던 철학자 Quine (콰인)의 업적을 기리는 의미에서 그의 이름으로 부른다. 콰인은 거의 모든 프로그래밍 언어에 대해서 존재하며, 하나의 프로그래밍 언어에 대해서도 여러가지가 존재할 수 있다. 관련 내용은 구글에서 Quine을 검색해보면 나온다.

3. 함수의 포인터에 관한 C의 문법 자체가 혼란스럽다. 이러한 혼란의 원인은 "함수는 식 안에서 '함수의 포인터'로 바뀌 읽힌다"라는 규칙 때문이다. 이에 대해 ANSI 규칙은 다음과 같이 기술하고 있다.

   (a) 함수는 식 안에서 '함수의 포인터'로 자동변환된다. 단 어드레스 연산자 &의 오퍼랜드일 때와 sizeof 연산자의 오퍼랜드일 때는 예외이다.

   (b) 함수 호출 연산자 () 는 함수가 아닌 함수의 포인터를 오퍼랜드로 한다.

   따라서 함수의 포인터에 대해 간접 참조 연산자 *를 적용하면 일단 '함수'가 되지만, 식 안이므로 즉시 함수의 포인터로 변환이 된다. 따라서 * 연산자를 함수의 포인터에 적용해도 아무 일이 일어나지 않는 것처럼 보이는 것이다.

   즉 문제에서, (*****printf) 는 결국 printf로 평가된다.

4. 앞의 문제와 마찬가지 이유로 함수의 포인터가 출력된다. 6 개의 출력문 모두 같은 값으로, 함수 function()의 포인터가 출력된다.

5. pb라는 포인터를 이용하여 배열 buf를 0이 아닌 1부터 어드레싱하기 위한 편법이다. 대부분의 C/C++ 프로그래머들이 배열을 0부터 시작한다고 알고 있으므로 추천할만한 방법은 아니고, 다만 포인터를 이렇게 사용할 수 있다는 걸 보여주는 예이다.

6. `#define` 매크로로 정의되어 있다. 따라서, 다양한 표현식이 들어갈 때, 잘못 해석될 수도 있다는 근본적인 문제가 있다. 따라서, 다음과 같이 바꾸어야 한다.

   ```
   #define SWAP(a,b) ((a)+=(b), (b)=(a)-(b), (a)-=(b))
   ```

   그럼에도 문제는 무조건 괄호를 넣는 방법으로 다양한 표현식이 들어가는 문제를 해결할 수 없다는 것이다. `i++` 와 같은 후위 증가 연산자를 넣을 경우 방법이 없다.

   게다가, 이 매크로의 경우 또 하나의 문제는, `a`나 `b`는 대입하는 객체인 `lvalue` 가 되기도 한다는 것이다. 그런 경우 반드시 `a`나 `b`는 변수이어야 하고 연산자로 결합된 표현식이면 안된다. 예를 들어, `i+j` 이나 `l-m` 과 같은 표현식에 `i+j=10` 이라는 식으로 대입을 할 수 없다. `i+j=10;`과 같은 명령문은 컴파일이 안된다. 따라서 `SWAP(i+j, 10)` 과 같은 명령은 분명히 컴파일 에러가 나지만, SWAP 매크로 자체에서는 에러를 발견하지 못한다. 즉, 매크로는 이러한 점들에 대해 미리 검사할 수 없다.

   (a) 인라인 함수를 쓰면 대부분의 문제가 해결된 거처럼 보인다. 그러나, 여전히 편법으로 swap을 했기 때문에 발생하는 문제는 남아있다. 예를 들어 이 경우의 문제는, 자기 자신과 교환을 할 경우 부작용이 발생한다는 것이다. 예를 들어, `swap(x,x)`는 x를 0으로 만들어 버린다.

   (b) `inline void swap(int& a, int& b) { if (a!=b) {a^=b; b^=a; a^=b;} }`

   (c)
   ```
   void swap(string& a, string& b)
   {
     a = a + b;
     b = a.substr(0,a.length()-b.length());
     a = a.substr(b.length());
   }
   ```

7. C/C++에서 배열 첨자 연산자 `[]`는 단순히 그 안의 내용을 포인터 연산한 거에 지나지 않는다. (다만 C++의 경우 배열 첨자 연산자 `[]`가 오버로딩되어 다른 의미가 될 수 있다는 점은 주의해야 할 것이다.) 예를 들어 `a[i]`는 실은 `*(a+i)`인 것이다.

   위의 규칙에 따라, `"Dongseo"[i]`는 `*("Dongseo"+i)`이며 문자열 `"Dongseo"`는 계산식에서 그 문자열있는 위치의 가장 처음 주소로 평가된다. 따라서, `i`가 증가함에 따라 한 글자씩 출력되어 Dongseo라고 찍히게 된다. 또한, `i["Dongseo"]`도 `*(i+"Dongseo")`로 해석되어, 위와 같은 결과를 내게 된다. 마지막으로, `i[nums]`도 `*(i+nums)`로 해석되어, 1234567이라고 찍히게 된다.

8. 1을 출력한다. 그 이유는 콤마 연산자(,)보다 대입 연산자(=)가 우선 순위가 낮기 때문이다. `money = 1,000,000` 에서 `money = 1`이 먼저 평가되면서 `money`에 1이 대입되고 그 결과로 `money = 1`라는 표현식은 1로 평가된다. 그 다음 콤마 연산자에 의해 `1,000`이 평가되어 0으로 평가되고, 그 다음 콤마 연산자에 의해 `0,000`은 최종적으로 0으로 평가된다. 그러나 `money`에는 앞에서 1로 대입되고 변하지 않는다.

1.1.3 단답형 (문제 2.3.3)

1. `if  sizeof`

2. (a) `void kick(char * name, int times=1);`

   (b) 수정할 필요 없다.

   (c) times가 디폴트 값을 가진다면 name에 대해 다음과 같이 "Chuck Norris"를 디폴트 값으로 가지게 할 수 있다.

   ```
   void kick(char * name = "Chuck Norris", int times=1);
   ```

   times가 디폴트 값을 가지지 않는다면 name와 times의 위치를 바꾸고, name에 대해 다음과 같이 "Chuck Norris"를 디폴트 값으로 가지게 할 수 있다. 이 경우, 정의에서도 위치를 바꾸어 주어야 한다.

   ```
   void kick(int times, char * name = "Chuck Norris");
   ```

3. 두 개의 전달 인자들 중에서 더 큰 것을 반환하는 함수 템플릿을 작성하라.

   ```
   template <typename T>
   const T& max(const T& t1, const T& t2)
   {
     return t1>t2 ? t1 : t2;
   }
   ```

   (a) `template const double& max<double>(const double&, const double&);`

   (b)
   ```
   template <> const Personnel& max<Personnel>(const Personnel&, const Personnel&);
   template <> const Personnel& max<Personnel>(const Personnel& t1, const Personnel& t2)
   {
     return t1.salary>t2.salary ? t1 : t2;
   }
   ```

4. const 제한자를 사용할 수도 있으나, 대부분의 경우 필요가 없다. const 제한자를 사용하는 이유는 전달되는 값이 변경되지 않게 하기 위해서다. 주로, 객체가 참조로 전달되는 경우, 전달된 객체가 변경되는 부작용을 없애고 싶다면, 이러한 제한자는 유용하다. 객체를 참조로 전달하는 이유는, 객체를 생성해서 복사하는 시간을 없애서 속도를 빠르게 하기 위해서다. 그러나, 기본 데이터 형은 복사되는 데 그리 많은 시간이 걸리지 않아서 참조로 전달되지 않는 경우가 많다. 따라서 기본 데이터 형은 주로 참조가 아니라 그대로 복사되서 전달되며, 원본은 변경되지 않는다. 따라서 이런 경우, const 제한자를 쓸 필요가 없다.

5. `void func(Student); // 값`
   `void func(Student&); // 참조`

1.2 프로그래밍 문제 숙제 (문제 2.4)

시간 관계상, 교재에 나온 문제들에 대한 해답은 생략하였다. 인터넷에서 다운받을 수 있는 모범 답안을 참조하기 바란다.

1.2.1 4 개의 말뚝을 가진 하노이의 탑 (문제 2.4.4)

3 개의 말뚝을 가지는 하노이의 탑은 다음과 같이 해결된다. 1 번 말뚝에 모든 디스크들이 있고, 2 번 말뚝으로 옮기고 싶어한다고 가정하자.

1. n-1 개를 1번에서 3번 말뚝에 옮긴다. 2번 말뚝을 이용한다.
2. 남은 1 개를 1번에서 2번 말뚝에 옮긴다.
3. n-1 개를 3번에서 2번 말뚝에 옮긴다. 1번 말뚝을 이용한다.

가장 간단한 해결책은 문제에서 서술하였듯이 위의 알고리즘을 다음과 같이 약간 변형하는 것이다.

1. n-2 개를 1번에서 4번 말뚝에 옮긴다. 2번과 3번 말뚝을 이용한다.
2. 1 개를 1번에서 3번 말뚝에 옮긴다.
3. 남은 1 개를 1번에서 2번 말뚝에 옮긴다.
4. 1 개를 3번에서 2번 말뚝에 옮긴다.
5. n-2 개를 4번에서 2번 말뚝에 옮긴다. 1번과 3번 말뚝을 이용한다.

그러나 다음과 같은 방법으로 훨씬 더 적은 횟수로 옮길 수 있다. 즉 n 개의 디스크를 옮기는 4 개의 말뚝을 가지는 하노이의 탑 알고리즘은 다음과 같다.

1. n 보다 작은 k 개를 1번에서 4번 말뚝에 옮긴다. 2번과 3번 말뚝을 이용한다.
2. 남은 n-k 개를 1번에서 2번 말뚝에 옮긴다. 3번 말뚝만 사용한다.
3. k 개를 4번에서 2번 말뚝에 옮긴다. 1번과 3번 말뚝을 이용한다.

이 때 k는 1보다 크거나 같고 n보다 작으며, 앞의 알고리즘을 사용했을 때 그 움직임의 수가 최소화되는 k이다. 본 해답에서는 하나 하나 직접 움직임의 수를 계산하여 최소화하는 k를 구했다.

위의 알고리즘에서 2번 문장을 보면 1번 말뚝에서 2번 말뚝으로 옮기되, 3번 말뚝만 사용한다는 걸 알 수 있다. 이미 4번 말뚝에는 앞에서 k 개를 옮겼기 때문이다. 즉 2번 문장은 3 개의 말뚝을 가지는 하노이의 탑 알고리즘으로 간단히 해결될 수 있다. 나머지 1번과 3번 문장은 다시 n 보다 작은 k 개의 디스크를 옮기는 4 개의 말뚝을 가지는 하노이의 탑 알고리즘이 재귀적으로 사용됨을 알 수 있다. 이를 구태여 알고리즘으로 쓰면 다음과 같다. `Hanoi3` 는 3 개의 말뚝을 가지는 하노이의 탑 알고리즘이며, $H_4(i)$ 는 4 개의 말뚝을 가지는 하노이의 탑 알고리즘이 i 개의 디스크를 옮기는 데 필요한 디스크를 이동한 전체 횟수(moves)이다.

위의 알고리즘에서 보듯이, 본 해답에서 제시하는 프로그램에서는 k 값을 구하기 위해 단순히 1에서 n-1까지의 모든 수에 대해 하나 하나 검사하여 가장 작은 수를 찾는 방법을 사용하였다.

이를 구현하기 위해서는 매우 큰 수의 계산이 요구되나 C++는 기본 라이브러리나 클래스 집합에 큰 수의 계산에 대한 라이브러리가 없다. 그래서, 위의 알고리즘은 자바로 구현하였다.

Hanoi4(from, to, using1, using2, n):
begin

1. $k = argmin_{1 \le i < n} \left(2H_4(i) + \left(2^{(n-i)} - 1\right)\right)$
2. Hanoi4(from, using2, to, using1, k) // $H_4(k)$
3. Hanoi3(from, to, using1, n-k) // $2^{(n-k)} - 1$
4. Hanoi4(using2, to, from, using1, k) // $H_4(k)$

end.

그림 1: n 개의 디스크를 옮기는 4 개의 말뚝을 가지는 하노이의 탑 알고리즘