객체지향 프로그래밍 노트
함수 포인터 (Function Pointer)

강대기*

2007년 10월 10일

제 1 절 정의 방법

함수 포인터를 함수를 가리키는 포인터이다.

함수 포인터를 정의하는 방법은 일반 포인터를 정의하는 방식과 비슷하다. 예를 들면 x라는 정수는 다음과 같이 정의한다.

```
int x;
```

정수를 가리키는 포인터 px은 다음과 같이 정의한다.

```
int *px;
```

이름 앞에 *만 붙었음을 알 수 있다.

함수는 어떻게 선언될까? 그건 바로 함수의 원형(prototype)이다. 즉 함수 포인터를 선언하는 방법은 근본적으로 함수의 원형(prototype)을 선언하는 방법과 동일하다. 다만, 함수의 원형에서 함수 이름 부분 앞에 *를 붙이면 된다.

예를 들면 다음이 정수를 반환하는 함수의 원형 선언일 경우,

```
int f(); // 함수
```

그 함수의 포인터는 다음과 같이 선언**되어야 할 것이다**. 그런데, 그렇지 않다.

```
int *pf(); // 정수의 포인터를 반환하는 함수
```

저것은 함수 포인터가 아니라 정수의 포인터를 반환하는 함수 선언이다. 이렇게 된 이유는 포인터를 위한 내용 참조 연산자인 '*'가 함수 호출을 위한 '()' 연산자보다 연산자 우선 순위가 낮기 때문이다. 해결책은 다음과 같이 괄호 묶기 연산자인 '()'로 포인터 연산자와 변수명을 둘러싸서 포인터 연산이 먼저 연결되도록 하는 것이다.

```
int (*pf)(); // 정수를 반환하는 함수의 포인터
```

이러한 방식은 비유하자면 마치 2+3*4라는 식에서 3*4가 먼저 계산되어 12가 되고 그 다음에 2가 더해져 14가 되는 것과 비슷하다. 연산자 우선 순위에서 *가 +보다 높기 때문이다. 따라서 2+3이 먼저 계산되게 하려면 (2+3)*4라는 식으로 괄호로 묶어줘야 하는 것이다.

다음은 함수와 그 함수에 대한 함수 포인터의 정의의 예이다.

- double func(); ⟶ double (*pfunc)();
- float func(int, long); ⟶ float (*pfunc)(int, long);
- double *func(int, long); ⟶ double *(*pfunc)(int, long);
- int *func(char *); ⟶ int *(*pfunc)(char *);

제 2 절 호출 방법

함수 포인터는 근본적으로 다음과 같이 호출한다.

```
(*fp)(); // 정수를 반환하는 함수의 포인터를 이용해서 그 함수를 호출
```

함수 포인터는 함수의 포인터이므로 함수의 주소값을 대입해야 한다. 이렇게 함수 포인터를 초기화할 때는 근본적으로 다음과 같이 한다.

```
int func()
{
  return 0;
}

int main()
{
  int (*fp1)() = &func;
  (*fp1)(); // 함수 포인터로 함수 호출

  // 또는
  int (*fp2)();
  fp2 = &func;
  (*fp2)(); // 함수 포인터로 함수 호출
}
```

이에 대한 간단한 예를 들어보자. 파이값 3.14에 함수 패러미터(전달인자)로 받은 정수를 곱해서 반환하는 함수의 포인터를 사용하는 예이다.

```
#include <iostream>

using namespace std;
```

```
double piX(int);

int main()
{
  double (*pp)(int) = NULL;
  pp = &piX; // pp = piX; 와 동일
  int x = 2;
  cout << "The result with " << x << " is " << (*pp)(2) << ".\n";
  return 0;
}

double piX(int x)
{
  cout << "We multiply pi with " << x << ".\n";
  return 3.14*(double)x;
}
```

함수 포인터로 pp를 선언했고 NULL로 일단 초기화했다. 또한 `pp = &piX;`로 piX 함수의 주소로 대입했다. 그런데, 함수의 주소 대신 다음과 같이 함수 자체로 대입해도 된다. 이에 대한 논의는 교재의 412 쪽의 NOTE에 나온다.

```
  pp = piX; // pp = &piX; 와 동일하다.
```

이를 더 이해하기 위해 다음의 프로그램을 보도록 하자.

```
#include <iostream>
using namespace std;

int add(int a, int b)
{
  return a+b;
}

int main()
{
  cout << "add(1,2) = " << add(1,2) << endl;

  int (*fp)(int,int);
  fp= &add; // fp = add 와 동일

  cout << "(*fp)(2,3) = " << (*fp)(2,3) << endl; // fp(2,3) 과 동일

  cout << "fp = "<< fp << endl;
  cout << "(&fp) = "<< (&fp) << endl;
  cout << "(*fp) = " << (*fp) << endl;

  cout << "add = " << add << endl;
```

```
  cout << "(&add) = " << (&add) << endl;
  cout << "(*add) = " << (*add) << endl;
}
```

이 프로그램을 실행하면 다음과 같은 결과를 낸다.

```
add(1,2) = 3
(*fp)(2,3) = 5
fp = 004110CD
(&fp) = 0012FF40
(*fp) = 004110CD
add = 004110CD
(&add) = 004110CD
(*add) = 004110CD
```

보듯이, 단순히 함수인 add에 대해서는 &add나 *add가 다 같은 결과(004110CD)를 냄을 알 수 있다. 따라서 fp = add와 fp = &add는 동일하다.

그런데 실행 결과를 보면, fp 와 (*fp)의 값도 동일함을 알 수 있다. 따라서, 위의 경우, fp(2,3) 과 (*fp)(2,3)도 같은 결과를 내는 것이다.

어떤 것을 쓸지는 각자의 편의에 따라 하면 되겠다. 필자의 경우, fp = &add와 (*fp)(2,3) 같은 좀 더 엄격한 방식을 선호한다.

제 3 절 사용 예

함수 포인터에 대한 다른 예로, 교재의 예에서 가져온 다음 예를 보도록 하자.

```
#include <iostream>

using namespace std;

double p1(int);
double p2(int);

// 두 번째 전달인자는 int형을 전달인자로 취하는
// double형 함수를 지시하는 포인터이다
void estimate(int, double (*)(int));

int main()
{
  cout << "Enter LOC: ";
  int code;
  cin >> code;
  cout << "Estimated time for P1:\n";
  estimate(code, &p1);
  cout << "Estimated time for P2:\n";
  estimate(code, &p2);
```

```
  return 0;
}

double p1(int lns)
{
  return 0.05 * lns;
}

double p2(int lns)
{
  return 0.03 * lns + 0.0004 * lns * lns;
}

void estimate(int lines, double (*pf)(int))
{
  cout << "For " << lines << " lines, "
    "it takes " << (*pf)(lines) << " time.\n";
}
```

여기서 estimate는 정수(int)와 함수 포인터를 받는 함수이다. estimate는 void로 아무 것도 반환하지 않는다. estimate가 두 번째 인자로 받는 함수 포인터는 정수 값을 받아서 double을 반환하는 함수의 포인터이다.

함수 포인터를 함수의 전달 인자로 사용하여 복잡한 switch문을 대신하는 예를 알아보겠다. 사칙연산을 하는 프로그램으로 우선 다음과 같은 프로그램을 보도록 하자.

```
#include <iostream>
using namespace std;

float plus(float a, float b) { return a+b; }
float minus(float a, float b) { return a-b; }
float multiply(float a, float b) { return a*b; }
float divide(float a, float b) { return a/b; }

void operate(float a, float b, char opCode)
{
   float result = 0;
   // execute operation
   switch(opCode)
   {
      case '+' : result = plus(a,b); break;
      case '-' : result = minus(a,b); break;
      case '*' : result = multiply(a,b); break;
      case '/' : result = divide(a,b); break;
   }
   cout << result << endl; // display result
```

```
}

int main()
{
   operate(2, 5, '+');
}
```

위의 switch 문은 함수 포인터를 사용하면 다음과 같이 바뀐다.

```
#include <iostream>
using namespace std;

float plus(float a, float b) { return a+b; }
float minus(float a, float b) { return a-b; }
float multiply(float a, float b) { return a*b; }
float divide(float a, float b) { return a/b; }

void operate(float a, float b, float (*op)(float,float))
{
  // 필자는 (*op)(a,b)를 더 선호하지만
  // 여기선 op(a,b)도 가능함을 보이기 위해 넣어봄
  float result = op(a,b);
  cout << result << endl; // display result
}

int main()
{
  // 필자는 &plus 를 더 선호하지만
  // 여기선 plus 도 가능함을 보이기 위해 넣어봄
  operate(2, 5, plus);
}
```

float (*op)(float,float)를 해석해 보자. 일단 op라는 변수 내지 상수가 있다. 연산자 우선 순위에 의해 (*op)가 먼저 결합되므로, 이 op는 근본적으로 포인터 변수임을 알 수 있다. 그 다음 함수 호출을 나타내는 (float, float)가 결합되므로, op는 float 와 float 두개의 인자를 가지는 함수의 포인터 변수인 것이다. 마지막으로 맨 앞의 float가 결합되므로, op는 float 값을 반환하는 float 와 float 두개의 인자를 가지는 함수의 포인터 변수이다.

제 4 절 복잡해 보이는 문제들

포인터가 그러하듯, 함수 포인터가 포인터나 다른 함수 포인터들과 결합되면 당장 보기엔 상당히 복잡하게 보일 수 있다. 전문가들이 짠 C나 C++ 프로그램들을 보면 다음과 같이 좀 복잡한 경우도 실제 있을 수 있다. 그러나 근본적으로 포인터나 함수 포인터가 선언되는 방법을 이해하고 차근차근 접근하면 다 이해할 수 있는 내용이다.

예를 들어 함수 포인터의 배열을 선언하는 방법을 보자.

```
double (*pf[3])(double, double)
```

해석해 보면, 일단 pf라는 변수 내지 상수가 있는 데, 배열을 나타내는 []이 포인터의 *보다 우선순위가 높으므로 pf[3]는 세 개의 원소를 가진 배열임을 알 수 있다. 그 다음에는 *이 붙어서 *pf[3]가 되므로, 포인터의 배열이 된다. 그 다음 (double, double)이 붙어서 *pf[3](double, double)은 double, double를 함수 인자로 가지는 함수 포인터의 배열이다. 마지막으로 맨 앞의 double이 붙어, 전체는 double을 반환하며 double, double를 함수 인자로 가지는 함수 포인터의 배열인 것이다.

다음은 함수의 원형과 그에 상응하는 함수 포인터 선언이다.

1. void operate(float, float, float (*)(float,float)) ⟶ void (*pOperate)(float, float, float (*)(float,float))
2. int (*func())() ⟶ int (*(*pFunc))()

1번 예는 앞에서 든 operate 함수인 void operate(float a, float b, float (*op)(float,float))의 원형과 그 원형에서 유래한 함수 포인터이다. 우리는 float (*op)(float,float) 가 float 값을 반환하는 float 와 float 두개의 인자를 가지는 함수의 포인터 변수의 선언이란 것을 알고 있다. pOperate는 아무 것도 반환하지 않으며, float, float 그리고 “float 값을 반환하는 float 와 float 두개의 인자를 가지는 함수의 포인터 형”을 세 개의 인자로 가지는 함수의 포인터인 것이다.

다소 복잡해 보이는 2번 예는 정수를 반환하는 함수의 포인터를 반환하는 함수인 func의 원형과 그 func에 대한 함수 포인터이다. 이에 대한 프로그램 예를 보도록 하자.

```
#include <iostream>
using namespace std;

int (*func())(); // function
int (*(*pFunc))(); // pointer

int hello()
{
  cout << "Hello\n";
  return 1;
}

int (*func())() { return &hello; }

int main()
{
  cout << ((*func))()() << endl;
  pFunc=&func;
  cout << ((*(*pFunc))())() << endl;
```

```
  return 0;
}
```

int (*func())(); 를 해석해 보자. 일단 func라는 변수 내지 상수가 있다. 포인터 간접 참조 연산자인 *와 함수 호출을 나타내는 () 연산자 중 ()가 우선 순위가 더 크므로, func()가 먼저 결합된다. 따라서 근본적으로 func는 함수이다. 그 다음 *이 결합되어 *func()가 되므로, func는 함수는 함수인데, '포인터를 변환하는 함수'가 된다. 그 다음 (*func())로 전체가 괄호로 둘러싸여진다. 그리고 나서 다시 함수 호출을 나타내는 ()가 붙어서 (*func())()가 되므로 , func는 함수의 포인터를 반환하는 함수인데, 특별히 전달받는 패러미터(인자)들은 없다. 마지막으로 맨 앞의 int가 붙어서, int (*func())() 전체는 **정수값을 반환하고 인자는 없는 함수의 포인터를 반환하는 함수**가 된다.

앞의 것을 이해했다면, **int (*(*pFunc)())();** 의 경우는 쉽다. **int (*func())();** 에서 func를 (*pFunc)로 바꾼 것에 불과하기 때문이다. 즉 pFunc는 포인터인데, **정수값을 반환하고 인자는 없는 함수의 포인터를 반환하는 함수의 포인터**인 것이다.

좀 더 쿨한 예로 '앤드류 쾨니히'의 'C 함정과 실수'에 나오는 다음의 예를 보자. ('앤드류 쾨니히'의 'C 함정과 실수'는 참고 문헌 목록에 있으며 전문적인 C 프로그래머가 되고자 하는 사람은 반드시 일독해야 할 책들 중 하나이다.)

```
int main()
{
  (*(void(*)())0)();
}
```

위와 같은 코드가 실제로 컴파일된다는 사실이 놀라울지도 모르겠지만, 실제 아무 에러 없이 컴파일된다. 물론 실행을 하면 에러가 날 것이다. 위의 코드가 컴파일이 된다는 사실은 문법적으로 문제가 없다는 것이다. 과연 위의 코드는 어떤 의미인가?

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.

여기까지 이해했다면, 이제 전부 이해가 되었을 것이다. 결과적으로 (*(void(*)())0)(); 는 0 번지에 들어있는 주소를 함수의 시작 번지로 삼아서 호출하는 (원래의 의도는 점프하는) 명령문이다.