Chapter 17

함수 포인터

함수 역시 메모리에 저장되며 시작 주소를 가진다.

따라서 함수를 가리킬 수 있는 함수 포인터가 제공되는 것은 당연한 일이다.

이번 장에서는 함수 포인터를 통해서 함수를 객체처럼 다루거나

더 효율적이고 강력하게 호출할 수 있는 방법을 살펴본다.

함수 포인터의 개요

1.1 함수 포인터의 개념

포인터는 어떤 대상(변수, 배열, 함수 등)을 가리키기 위해 필요한 정보를 담고 있는 객체(변수)를 말한다. 지금까지는 변수나 배열을 가리키는 포인터만을 다루었지만 이번 장은 함수를 가리키는 포인터를 다루게 된다. 일명 '함수 포인터'이다.

포인터는 특정한 메모리 영역을 가리키고, 그 영역을 해석할 방법을 제공하기 위하여 일반적으로 주소와 타입을 필요로 한다. 함수 포인터 역시 함수의 주소와 함께 함수의 타입이라고 할 수 있는 시그니처 정보를 포함한다.

C 언어는 클래스가 없어서 멤버 함수라는 개념이 존재하지 않는다. 즉, C 언어의 함수는 전역 함수만 존재한다. 따라서 함수 포인터 역시 전역 함수만을 가리키는 포인터로 무척 간단했다. 그러나 C++의 경우 클래스와 함께 멤버 함수가 도입되면서 함수 포인터가 상당히 복잡해지기 시작한다. C++ 클래스의 상속은 다른 언어와는 다르게 다중 상속을 비롯하여 가상 상속이 가능하므로 이에 따른 멤버 함수 호출 처리가 복잡해지면서 함수 포인터 역시 같이 복잡해진 측면이 있다. 사실 제대로 함수 포인터를 다루려면 방대한 분량의 지면이 필요하며 그 복잡도가 상당하지만 이 책에서는 기본적인 함수 포인터의 소개 및 사용법 위주로 살펴보고자 한다.

1.2 함수 포인터 타입 파생 규칙

앞서 '타입' 장에서 타입 파생 규칙 대해서 설명했다. 기준 타입에서 배열, 포인터, 함수 등으로 타입을 파생시키는 규칙이다. 여기서 필요한 파생 타입은 바로 포인터 타입 파생과 함수 타입 파생이다. 예전 내용을 복습한다는 생각으로 다시 한번 살펴보자.

- **포인터 타입 파생:** 포인터가 가리키는 타입 T를 기준으로 포인터 타입을 파생하려면 타입 T의 이름 자리 [NP] 바로 앞에 간접(*) 연산자를 붙인다. 간접(*) 연산자의 결합 순위가 떨어질 경우 간접(*) 연산자와 이름 자리 [NP]를 괄호로 묶는다. 파생된 포인터 타입의 이름 자리는 그대로 유지된다.
- **함수 타입 파생:** 함수는 반환 타입 T를 기준으로 이름 자리 [NP] 바로 뒤에 (인자 리스트)를 붙인다. 파생된 함수 타입의 이름 자리는 그대로 유지된다.

주목해야 할 점은 포인터가 가리키는 타입 T가 함수 타입이라는 점이다. 함수 타입 역시 파생된 타입이고 이름 자리 [NP] 바로 뒤에 (인자 리스트)가 붙어 있다. 여기서 중요한 점은 간접(*) 연산자와 이름 자리 [NP]의 결합 순위는 (인자 리스트)와 이름 자리 [NP]의 결합 순위보다 낮으므로 함수 포인터 타입을 파생할 경우에는 반드시 간접(*) 연산자와 이름 자리 [NP]를 괄호로 묶어야 한다는 사실이다.

함수 타입 및 포인터 타입 파생 규칙을 이용하여 실제 함수 포인터 타입을 만들어보고, 반환 타입이 int이고 인자 타입이 double인 함수를 가리키는 포인터 타입을 파생시켜보자.

1. 반환 타입 int를 기준으로 한다. → int [NP]
2. 이름 자리 [NP] 바로 뒤에 (인자 리스트)를 붙인다. → int [NP](double)
3. 간접(*) 연산자를 이름 자리 [NP] 바로 앞에 붙인다. 결합 우선순위를 고려하여 간접(*) 연산자와 이름 자리 [NP]를 괄호로 묶는다. → int (*[NP])(double)
4. [NP]를 생략하여 최종 타입 표기를 한다. → int (*)(double)

int (*)(double)은 참 기묘하게 생긴 타입임을 알 수 있다. 이처럼 대부분 함수 포인터 타입은 복잡한 형식을 가지게 된다. 그렇다면 이 타입을 통해서 실제 함수 포인터를 정의하려면 어떻게 할까?

> **파생 타입 객체 정의:** 타입의 이름 자리 [NP]에 객체의 이름을 써주는 것이 끝이다(정의 구문이므로 끝에 세미콜론(;)을 붙인다. 함수인 경우 함수 본체를 정의해야 하므로 블록을 추가한다).

위의 규칙에 따라서 int (*)(double)의 객체(함수 포인터) p를 정의하는 방법은 다음과 같다.

- int (*[NP])(double)
- 이름 자리 [NP] 대신에 객체 이름 p를 쓰고 세미콜론을 붙인다. → int (*p)(double);

따라서 p는 인자 타입이 double이고 반환 타입이 int인 함수를 가리키는 포인터이다.

전역 함수 포인터

전역 함수 혹은 클래스 정적 함수를 가리키는 포인터이다. C 언어는 전역 함수만 있었기 때문에 C 언어의 함수 포인터와 같다.

[예제 17-1] 전역 함수 포인터 사용

```
1  #include <iostream>
2  using namespace std;
3
4  void Func()
5  {
6     cout << "Func" << endl;
7  }
8
9  void main()
10 {
11    void (*pf1)();
12    pf1 = &Func;
13
14    void (*pf2)() = &Func;
15    void (*pf3)() = Func;
16
17    pf1();
18    pf2();
19    pf3();
20    (*pf1)();
21    (*pf2)();
22    (*pf3)();
23 }
```

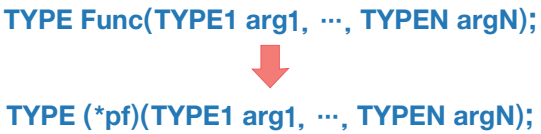
[예제 17-1]은 그리 길지 않지만 설명할 것은 상당히 많은 코드이다. 4행에는 전역 함수 Func가 있다. Func를 가리키는 함수 포인터 pf1, pf2, pf3을 정의하고, 함수 포인터를 통해서 함수를 호출하는 방법을 보여준다.

2.1 전역 함수 포인터 정의

먼저 Func와 같은 타입의 함수(인자열이 없고 반환 타입이 void인 함수)를 가리키는 전역 함수 포인터 pf를 정의하는 과정을 살펴보자.

1. 반환 타입 void를 기준으로 한다. → void [NP]
2. 이름 자리 [NP] 바로 뒤에 (인자 리스트)를 붙인다. → void [NP]( )
3. 간접(*) 연산자를 이름 자리 [NP] 바로 앞에 붙인다. 결합 우선순위를 고려하여 간접(*) 연산자와 이름 자리 [NP]를 괄호로 묶는다. → void (*[NP])( )
4. 이름 자리 [NP] 대신에 객체 이름 pf를 쓰고 세미콜론을 붙인다. → void (*pf)( );

위의 과정은 정석을 따른 것이고 실제로는 간단하게 전역 함수 포인터를 정의할 수 있다.



[그림 17-1] 전역 함수 포인터 정의

[그림 17-1]과 같이 함수의 선언에서 함수명을 (*포인터명)으로 대체하면 쉽게 함수 포인터를 정의할 수 있다. 그림에서 pf는 함수 Func와 같은 타입의 함수를 가리킬 수 있는 함수 포인터이다. 그림과 같은 단축 방식을 사용하여 pf를 정의해보자.

1. 함수를 선언한다. → void Func( );
2. 함수명을 (*포인터명)으로 대체한다. → void (*pf)( );

pf를 정의하는 방법을 사용하여 똑같이 pf1, pf2, pf3을 쉽게 정의할 수 있다.

2.2 전역 함수 포인터 대입

함수 포인터 pf1, pf2, pf3은 아직 실제 함수를 가리키고 있는 것은 아니다. 단지 인자열이 없고, 반환 타입이 void인 어떤 함수라도 가리킬 수 있는 포인터를 의미한다. 함수 포인터가 실제 함수를 가리키도록 하려면 함수를 대입해야 한다. 함수를 대입하는 형식 세 가지를 살펴보자.

11행에서 이미 함수 포인터 pf1이 정의되었다. 12행에서 pf1이 함수 Func를 가리키도록 설정하는데 pf1에 &Func를 대입한다. 여기서 &Func는 함수 Func를 피연산자로 주소(&) 연산자를 적용한 것으로 피연산자(함수)의 주소를 나타낸다. 11행, 12행은 함수 포인터의 정의와 대입을 각각 나눠서 하였지만 정의와 대입을 한번에 할 수도 있다. 14행, 15행이 바로 함수 포인터의 정의 및 대입을 보여준다.

15행에서 함수 포인터 pf3에 함수 Func를 대입한다. 정석대로라면 &Func를 대입해야 할 것이다. 그러나 전역 함수에 한해서는 함수 이름만 써줘도 상관없다. 전통적으로 C 언어에서 함수 이름은 곧 함수 주소를 나타내기 때문이다. 참고로 멤버 함수의 경우 반드시 주소(&) 연산자를 사용해야 한다. 따라서 이 책은 통일된 방식으로 함수를 함수 포인터에 대입할 경우 항상 주소(&) 연산자를 사용할 것이다.

2.3 전역 함수 포인터 호출

이제 함수 포인터를 통해서 함수를 호출해보자. 17~22행이 함수 포인터를 이용한 함수 호출 방법을 보여준다. 17~19행은 함수 포인터를 마치 함수처럼 사용하는 방법을 보여준다. 함수 포인터와 함수 이름은 곧 함수의 주소를 나타내므로 서로 호환될 수 있다.

20~22행은 함수 포인터의 정석 호출 방법이다. 간접(*) 연산자를 포인터에 적용하면 포인터가 가리키는 대상을 나타낸다. 즉, 함수 포인터에 간접(*) 연산자를 적용하여 함수를 나타내도록 한 것이다. 즉, (*pf1), (*pf2), (*pf3)은 Func와 같다. 여기서 왜 간접(*) 연산자와 함수 포인터에 괄호를 했을까? 간접(*) 연산자의 결합 우선순위가 낮기 때문이다.

2.4 전역 함수 포인터 타입 재정의

[예제 17-2] 함수 포인터 열거 정의

```
1  void main()
2  {
3     int a1, a2, a3;    // OK
4     void (*pf1, *pf2, *pf3)();    // Error
5  }
```

[예제 17-2]의 3행은 int 변수 a1, a2, a3을 열거해서 정의한다. 콤마(,) 연산자를 이용하여 변수를 열거하면서 정의하는 것은 코드의 감소 및 가독성을 높인다. 함수 포인터의 경우 타입 형식이 복잡하므로 오히려 열거 정의가 더욱 절실히 필요하다. 그러나 4행과 같은 함수 포인터 열거 정의는 지원되지 않기 때문에 오류가 발생한다. 대신 이 문제를 해결할 수 있는 좋은 방법이 있긴 하다. 바로 typedef를 사용하는 것이다. typedef를 재정의하면 다음과 같다.

❶ 원하는 타입을 표기한 후에 이름 자리 [NP]를 새로운 타입 이름으로 교체한다.

❷ 맨 앞에 typedef를 붙이고 마지막에는 세미콜론(;)을 붙인다.

typedef의 장점은 새로 재정의된 타입의 이름 자리 [NP]는 원시 타입처럼 타입 키워드 바로 뒤에 위치한다는 점이다. 즉, typedef를 사용하여 함수 포인터 타입을 재정의할 경우 원시 타입처럼 열거식으로 정의를 할 수 있다.

[예제 17-3] typedef 함수 포인터 타입 재정의

```
1  #include <iostream>
2  using namespace std;
3
4  void Func()
5  {
6     cout << "Func" << endl;
7  }
8
9  typedef void (*PFUNC)();
10
11 void main()
12 {
13    PFUNC pf1, pf2, pf3;
14
15    pf1 = pf2 = pf3 = &Func;
16
17    pf1();
18    pf2();
19    pf3();
20 }
```

[예제 17-3]은 typedef를 이용하여 함수 포인터 타입을 재정의하였다. 9행을 살펴보자. PFUNC가 바로 함수 포인터 타입을 나타낸다. PFUNC가 유도된 과정은 다음과 같다.

❶ 재정의하길 원하는 타입을 표기한다. → void (*[NP])()

❷ 이름 자리 [NP]를 새로운 타입 이름으로 교체한다. → void (*PFUNC)()

❸ 맨 앞에 typedef를 붙이고 마지막에 세미콜론(;)을 붙인다. → typedef void (*PFUNC)();

13행에서 볼 수 있듯이 typedef로 새로 정의된 타입 PFUNC는 원시 타입처럼 객체(변수)를 열거식으로 정의할 수 있다.

2.5 전역 함수 포인터와 타입 관계

함수의 중복 정의에서 인자 타입열에 대해서 설명했다. 인자 타입열이란 각 인자의 타입이 순서대로 배치된 순열을 의미한다. 인자의 개수가 같고, 인자의 타입이 나열된 순서도 같을 때 인자열이 같다고 한다.

전역 함수의 타입을 구성하는 요소는 반환 타입과 인자 타입열이 주가 된다(물론 함수 호출 규약과 const 한정 여부 등도 함수 타입의 구성 요소이긴 하다). 따라서 두 전역 함수가 있을 때 반환 타입도 서로 같고, 인자 타입열도 일치한다면 두 전역 함수는 서로 같은 타입이라고 말할 수 있다(역시 기타 요소도 동일한 경우를 가정한다). 함수 포인터는 대상 타입(가리키는 함수의 타입)과 같은 타입의 함수라면 어떤 것도 가리킬 수 있다. 반대로 타입이 다른 함수를 함수 포인터에 대입할 수 없다.

[예제 17-4] 함수 포인터 대상 타입과 함수의 타입 일치

```
1  #include <iostream>
2  using namespace std;
3
4  int Square(int a)
5  {
6     return a * a;
7  }
8
9  int Add(int a, int b)
```

```
{
   return a + b;
}

int Subtract(int a, int b)
{
   return a - b;
}

typedef int (*PFUNC1)(int);
typedef int (*PFUNC2)(int, int);

void main()
{
   PFUNC1 pf1;
   PFUNC2 pf2;

   pf1 = &Square;
   cout << pf1(1) << endl;     // 1 출력

   pf2 = &Add;
   cout << pf2(1, 1) << endl;  // 2 출력

   pf2 = &Subtract;
   cout << pf2(4, 1) << endl;  // 3 출력

   //  pf1 = &Add;   // Error
   //  pf2 = &Square;  // Error
}
```

```
1
2
3
```

[예제 17-4]는 함수 타입과 함수 포인터의 대상 타입의 일치가 중요함을 보여준다. 실제 함수 타입과 함수 포인터의 대상 타입을 표로 정리해보자.

[표 17-1] 함수 타입과 함수 포인터 대상 타입

함수	함수 타입
Square	int (int)
Add	int (int, int)
Subtract	int (int, int)

함수 포인터	함수 포인터 타입	대상 타입
pf1	PFUNC1	int (int)
pf2	PFUNC2	int (int, int)

[표 17-1]에서 중요한 것은 함수 타입과 대상 타입이 일치할 때만 함수를 함수 포인터에 대입할 수 있다는 점이다. 따라서 27행처럼 pf1에는 Square 함수만 대입 가능하고, 30행, 33행처럼 pf2에는 Add와 Subtract 함수가 대입 가능하다. 36행, 37행처럼 타입이 일치하지 않을 경우에는 컴파일 오류가 발생한다.

3 멤버 함수 포인터

멤버 함수 포인터는 클래스의 멤버 함수를 가리킨다. 전역 함수 포인터처럼 포인터를 정의하고 멤버 함수를 대입한 뒤에 호출하는 과정을 거친다.

[예제 17-5] 멤버 함수 포인터 사용

```
1  #include <iostream>
2  using namespace std;
3
4  class CTest
5  {
6  public:
7     void Func()
8     {
9        cout << "Func" << endl;
10    }
11 };
```

```
12
13 void main()
14 {
15    void (CTest::*pf1)();
16    pf1 = &CTest::Func;
17
18    void (CTest::*pf2)() = &CTest::Func;
19
20    CTest t;
21    (t.*pf1)();
22    (t.*pf2)();
23
24    CTest* pT = &t;
25    (pT->*pf1)();
26    (pT->*pf2)();
27 }
```

```
Func
Func
Func
Func
```

[예제 17-5]의 클래스 CTest는 멤버 함수 Func를 포함한다. 멤버 함수를 가리킬 수 있는 멤버 함수 포인터 pf1, pf2를 정의하여 Func를 대입한 뒤에 pf1, pf2를 이용하여 함수 호출하는 방법을 보여준다.

3.1 멤버 함수 포인터 정의

먼저 CTest::Func와 같은 타입의 함수(인자열이 없고 반환 타입이 void인 CTest의 멤버 함수)를 가리키는 함수 포인터 pf를 정의하는 과정을 살펴보자. 전역 함수 포인터 정의와 다른 점은 멤버 함수이기 때문에 클래스와 범위 연산자를 고려해야 한다는 점이다.

❶ 반환 타입 void를 기준으로 한다. → void [NP]

❷ 이름 자리 [NP] 바로 뒤에 (인자 리스트)를 붙인다. → void [NP]( )

❸ 멤버 함수가 되도록 클래스와 범위 연산자를 [NP] 바로 앞에 붙인다.

→ void CTest::[NP]( )

❹ 간접(*) 연산자를 이름 자리 [NP] 바로 앞에 붙인다. 결합 우선순위를 고려하여 클래스부터 이름 자리 [NP]까지 괄호로 묶는다. → void (CTest::*[NP])( )

❺ 이름 자리 [NP] 대신에 객체 이름 pf를 쓰고 세미콜론을 붙인다. → void (CTest::*pf)( );

위의 과정은 정석을 따른 것이고 실제로는 간단하게 멤버 함수 포인터를 정의할 수 있다.

TYPE ClassName::Func(TYPE1 arg1, ···, TYPEN argN);

TYPE (ClassName*pf)(TYPE1 arg1, ···, TYPEN argN);

[그림 17-2] 멤버 함수 포인터 정의

[그림 17-2]와 같이 함수의 선언에서 함수명을 *포인터명으로 대체한 후에 클래스명부터 포인터명까지 괄호로 묶으면 쉽게 멤버 함수 포인터를 정의할 수 있다. 그림에서 pf는 CTest::Func와 같은 타입의 함수를 가리킬 수 있는 멤버 함수 포인터이다. 그림과 같은 단축 방식을 사용하여 pf를 정의해보자. pf를 정의하는 방법을 사용하여 똑같이 pf1, pf2를 쉽게 정의할 수 있다.

❶ 함수를 선언한다. → void CTest::Func( );

❷ 함수명을 *포인터명으로 대체하고 클래스부터 포인터명까지 괄호로 묶는다.
→ void (CTest::*pf)( );

3.2 멤버 함수 포인터 대입

멤버 함수 포인터 pf1, pf2가 아직 실제 함수를 가리키고 있는 것은 아니다. 단지 인자열이 없고, 반환 타입이 void인 CTest의 어떤 멤버 함수라도 가리킬 수 있는 포인터를 의미한다. 멤버 함수 포인터가 실제 함수를 가리키도록 하려면 해당 클래스의 멤버 함수를 대입해야 한다. 함수를 대입하는 형식 두 가지를 살펴보자.

15행에서 이미 멤버 함수 포인터 pf1이 정의되었다. 16행에서 pf1이 CTest::Func를 가리키도록 pf1에 &CTest::Func를 대입한다. 여기서 &CTest::Func는 멤버 함수 Func를 피연산자로 주소(&) 연산자를 적용한 것으로 피연산자(함수)의 주소를 나타낸다. 15행, 16행은 멤버 함수 포인터의 정의와 대입을 각각 나눠서 하였지만 정의와 대입을 한번에 할 수도 있다. 18행

이 바로 멤버 함수 포인터의 정의 및 대입을 보여준다. 여기서 전역 함수 포인터와의 차이점이 있는데, 전역 함수를 대입할 때는 주소(&) 연산자를 사용하지 않아도 함수 대입이 되었지만 멤버 함수의 경우 반드시 주소(&) 연산자를 사용해야 한다.

3.3 멤버 함수 포인터 호출

이제 멤버 함수 포인터를 통해서 함수를 호출해보자. 멤버 함수를 호출하기 위해서는 해당 멤버 함수를 포함하는 클래스의 객체가 반드시 정의되어 있어야 한다. 멤버 함수 호출 방식이 내부적으로 기준 객체를 내부 인자로 전달하기 때문이다. 멤버 함수 안에서는 this 키워드를 통해서 전달된 기준 객체에 접근할 수 있다.

20행에서는 CTest 객체 t를 정의했다. 24행에서는 CTest 포인터 pT를 정의하고 t를 가리키도록 하였다. 21행, 22행은 t를 기준으로 멤버 함수 포인터 pf1, pf2를 이용하여 함수를 호출한다. 생소하게 보일 수 있는데, 직접 멤버(.) 연산자와 간접(*) 연산자를 혼합하여 사용하였다. 먼저 간접(*) 연산자가 함수 포인터에 적용되어 함수 포인터가 가리키는 함수를 나타내고, 직접 멤버(.) 연산자를 이용하여 t를 기준으로 호출한다.

25행, 26행은 포인터 pT를 기준으로 멤버 함수 포인터 pf1, pf2를 이용하여 함수를 호출한다. 역시 생소한 것은 마찬가지인데, 이번에는 간접 멤버(->) 연산자와 간접(*) 연산자를 혼합하여 사용하였다. 간접 멤버(->) 연산자가 사용된 이유는 pT가 포인터이기 때문이다.

멤버 함수 포인터 호출에서 주의할 점은 인자열의 결합 우선순위가 높기 때문에 반드시 기준 객체와 함수 포인터까지 괄호로 묶어야 한다는 점이다.

3.4 멤버 함수 포인터 타입 재정의

멤버 함수 포인터 타입 역시 typedef를 이용하여 재정의가 가능하다.

[예제 17-6] typedef 멤버 함수 포인터 타입 재정의

```
1  #include <iostream>
2  using namespace std;
3
```

```
4  class CTest
5  {
6  public:
7     void Func()
8     {
9        cout << "Func" << endl;
10    }
11 };
12
13 typedef void (CTest::*PFUNC)();
14
15 void main()
16 {
17    PFUNC pf1, pf2, pf3;
18    pf1 = pf2 = pf3 = &CTest::Func;
19
20    CTest t;
21    (t.*pf1)();
22    (t.*pf2)();
23    (t.*pf3)();
24
25    CTest* pT = &t;
26    (pT->*pf1)();
27    (pT->*pf2)();
28    (pT->*pf3)();
29 }
```

[예제 17-6]은 typedef를 이용하여 멤버 함수 포인터 타입을 재정의하였다. 13행의 PFUNC가 바로 멤버 함수 포인터 타입을 나타낸다. PFUNC가 유도된 과정은 다음과 같다.

❶ 재정의하길 원하는 타입을 표기한다. → void (CTest::*[NP])()

❷ 이름 자리 [NP]를 새로운 타입 이름으로 교체한다. → void (CTest::*PFUNC)()

❸ 맨 앞에 typedef를 붙이고 마지막에 세미콜론(;)을 붙인다.

→ typedef void (CTest::*PFUNC)();

17행에서 볼 수 있듯이 typedef로 새로 정의된 타입 PFUNC는 원시 타입처럼 객체(변수)를 열거식으로 정의할 수 있다.

3.5 멤버 함수 포인터와 타입 관계

멤버 함수의 타입을 구성하는 요소는 전역 함수와 마찬가지로 반환 타입과 인자 타입열이 주가 된다. 주의할 점은 멤버 함수를 포함하는 클래스도 주요 요소가 된다는 것이다. 따라서 서로 다른 클래스의 반환 타입과 인자 타입열이 일치하는 멤버 함수가 있어도 각각 속하는 클래스가 다르기 때문에 함수의 타입은 다른 것으로 인정된다.

멤버 함수 포인터도 대상 타입(가리키는 객체의 타입)이 함수의 타입과 일치하는 함수라면 어떤 것도 가리킬 수 있다. 반대로 타입이 다른 함수를 함수 포인터에 대입할 수 없다.

[예제 17-7] 함수 포인터 대상 타입과 함수의 타입 일치

```
1  #include <iostream>
2  using namespace std;
3
4  class CA
5  {
6  public:
7     void Func1()
8     {
9        cout << "CA::Func1" << endl;
10    }
11
12    void Func2()
13    {
14       cout << "CA::Func2" << endl;
15    }
16 };
17
18 class CB
19 {
```

```
20  public:
21     void Func1()
22     {
23        cout << "CB::Func1" << endl;
24     }
25
26     void Func2()
27     {
28        cout << "CB::Func2" << endl;
29     }
30  };
31
32  typedef void (CA::*PAFUNC)();
33  typedef void (CB::*PBFUNC)();
34
35  void main()
36  {
37     PAFUNC pfa[2];
38     pfa[0] = &CA::Func1;   // OK
39     pfa[1] = &CA::Func2;   // OK
40
41     PBFUNC pfb[2];
42     pfb[0] = &CB::Func1;   // OK
43     pfb[1] = &CB::Func2;   // OK
44
45     pfa[0] = &CB::Func1;   // Error
46     pfa[1] = &CB::Func2;   // Error
47     pfb[0] = &CA::Func1;   // Error
48     pfb[1] = &CA::Func2;  // Error
49  }
```

[예제 17-7]은 두 개의 클래스 CA, CB가 등장한다. 각 클래스는 멤버 함수 Func1, Func2를 포함한다. CA::Func1, CA::Func2는 같은 클래스의 멤버 함수이고, 반환 타입과 인자 타입 열이 동일하므로 같은 타입의 함수로 취급된다. 이 타입의 함수를 가리킬 수 있는 함수 포인터

타입은 32행의 PAFUNC이다. 마찬가지로 CB::Func1, CB::Func2도 같은 타입의 함수이며 33행의 PBFUNC가 함수 포인터 타입이 된다. 그러나 주의할 점이 있는데, CA::Func1와 CB::Func1은 반환 타입과 인자 타입열이 동일하지만 포함되는 클래스가 서로 다르므로 다른 타입이 된다.

37행, 38행은 PAFUNC와 PBFUNC를 통해서 배열을 정의했다. 37행의 pfa 배열의 요소는 오직 CA::Func1, CA::Func2만을 대입할 수 있다. 반대로 41행의 pfb 배열의 요소는 CB::Func1, CB::Func2가 대입 가능하다. 그 외에 45~48행처럼 서로 타입이 다른 경우는 오류로 처리된다.

4 함수 포인터의 활용

지금까지 함수 포인터를 살펴보았는데 이것을 언제 사용하는지 의문이 들 수 있다. 단순하게 함수를 호출하는 것이 더 쉬운데 굳이 포인터에 함수를 대입하고 호출하는 과정이 번거로울 수 있기 때문이다. 함수 포인터는 함수를 객체화한다는 점에서 활용도가 크다. 객체는 함수의 인자로 전달할 수도 있으며, 반환 값으로 사용할 수도 있기 때문이다. 즉, 함수 자체를 전달할 수 있다는 의미이다. 실제로 비슷한 용도로 함수 포인터가 많이 사용된다. 여기서는 표준 C 라이브러리가 제공하는 정렬 함수인 qsort를 사용하면서 함수 포인터가 활용되는 모습을 살펴보겠다.

```
void qsort(void* base, size_t num, size_t width,
   int (*compare)(const void*, const void*));
```

qsort는 정렬을 수행하는 함수인데 입력으로 배열을 전달하면 배열 각 요소들을 퀵 정렬Quick Sort을 이용하여 정렬한 후 돌려주는 함수이다. 각 인자들을 설명하면 base는 배열을 나타내고, num은 배열 요소의 개수, width는 요소의 크기이다.

가장 중요한 인자가 바로 마지막 인자인 compare인데 이것이 바로 '함수 포인터'이다. compare는 형태로 보았을 때, const void* 타입 인자 두 개를 받고 반환 타입이 int인 함수를 가리키는 함수 포인터이다. 왜 compare가 필요할까? 두 개의 요소를 정렬하기 위해서는 정렬 기준이 필요하다. 즉, 두 요소 사이의 대소 관계를 정할 기준을 말한다. 숫자의 경우 대소 관계가 확

실하지만 사용자 정의 자료형의 경우 기준은 정하기 나름이 된다. 가령 사람 사이의 대소 기준을 정한다고 할 경우 나이, 이름, 키, 몸무게 등으로 기준을 각각 다르게 할 수 있다. 즉, 대소 기준이 되는 함수를 마련하고 그 함수를 가리키는 함수 포인터 compare를 전달하면 해당 함수의 대소 기준으로 퀵 정렬을 하는 것이 핵심이다.

[예제 17-8] 함수 포인터를 이용하는 qsort

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

class CPerson
{
public:
   char* m_Name;
   int m_Age;
};

int CompareAge(const void* a, const void* b)
{
   CPerson* pa = (CPerson*)a;
   CPerson* pb = (CPerson*)b;

   if(pa->m_Age < pb->m_Age)
   {
      return -1;
   }
   else if(pa->m_Age > pb->m_Age)
   {
      return 1;
   }

   return 0;
}

void main()
```

```
29  {
30     const void FamilyCount = 5;
31
32     CPerson arr[FamilyCount] =
33     {
34        { "Grand Father", 70},
35        { "Father", 42},
36        { "Child", 8 },
37        { "Grand Mother", 67},
38        { "Mother", 38}
39     };
40
41     qsort(arr, FamilyCount, sizeof(CPerson), &CompareAge);
42
43     for(int i = 0; i < FamilyCount; i++)
44     {
45        cout << arr[i].m_Name << " " << arr[i].m_Age << endl;
46     }
47  }
```

```
Child 8
Mother 38
Father 42
Grand Mother 67
Grand Father 70
```

[예제 17-8]은 함수 포인터를 전달하는 qsort 사용법을 보여준다. qsort로 정렬할 대상은 클래스 CPerson 객체로 한다. 중요한 부분은 비교 함수인 11행의 CompareAge이다. 두 개의 인자는 각각 CPerson 객체의 포인터이다. 따라서 13행, 14행처럼 CPerson*으로 타입 변환을 해서 사용한다. 비교 함수는 a, b의 대소 관계에 따라 음수, 0, 양수로 반환을 해야 한다. CompareAge는 인자로 전달된 CPerson의 멤버 데이터인 m_Age를 대소 기준으로 삼았다.

이제 main을 살펴보자. 32행에서 CPerson 배열 arr을 정렬되지 않은 상태로 구성한 뒤에 41행에서 qsort를 호출한다. qsort에 전달되는 인자는 배열과 배열의 요소 수, 요소의 크기, 그리

고 제일 중요한 비교 함수 포인터로 &CompareAge이다. 정렬이 완료된 뒤에 43행에서는 for 문을 통해서 배열의 요소를 출력한다. 결과를 확인하면 나이 순으로 정렬된 것을 확인할 수 있다.

함수 포인터의 크기

포인터의 크기는 어떻게 정해질까? 일반적으로 시스템에 따라서 포인터 크기는 고정되는 것으로 알려져 있다. 보통 32비트 x86에서 4바이트, 64비트 x64에서 8바이트로 알고 있다. 그러나 엄밀히 말해서 이것은 주소를 담는 레지스터의 크기를 말하는 것이지 포인터의 크기는 아니다. 포인터는 메모리의 특정 영역을 차지하는 대상을 가리키는 것으로서 당연히 해당 영역의 주소 정보도 포함하지만 그 외에 해당 영역을 해석하기 위한 부가 정보도 포함하는 새로운 객체이기 때문이다. 따라서 주소 이외의 정보가 있을 경우 포인터의 크기는 더 커질 수도 있다. 실제로 이것을 확인할 수 있는 포인터가 바로 함수 포인터이다.

[예제 17-9] 함수 포인터의 크기

```
1  #include <iostream>
2  using namespace std;
3
4  class CTestA
5  {
6  };
7
8  class CTestB : public CTestA
9  {
10 };
11
12 class CTestC : public CTestA
13 {
14    virtual ~CTestC();
15 };
16
17 class CTestD : public CTestA, public CTestB
```

```
18  {
19  };
20
21  void (*gpf)();
22  void (CTestA::*mpfa)();
23  void (CTestB::*mpfb)();
24  void (CTestC::*mpfc)();
25  void (CTestD::*mpfd)();
26
27  void main()
28  {
29     cout << "Global: " << sizeof(gpf) << endl;
30     cout << "CTestA: " << sizeof(mpfa) << endl;
31     cout << "CTestB: " << sizeof(mpfb) << endl;
32     cout << "CTestC: " << sizeof(mpfc) << endl;
33     cout << "CTestD: " << sizeof(mpfd) << endl;
34  }
```

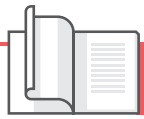
```
Global: 8
CTestA: 8
CTestB: 8
CTestC: 16
CTestD: 16
```

[예제 17-9]는 다양한 종류의 함수 포인터의 크기를 출력한다. 전역 함수 포인터부터 멤버 함수 포인터의 크기도 출력하는데, 특히 멤버 함수의 경우 클래스의 구조에 따라 멤버 함수 포인터의 크기가 달라지는 것을 확인할 수 있다. 특히 자식 클래스이면서 처음으로 가상 함수를 선언한 CTestC와 다중 상속을 하는 CTestD의 경우 멤버 함수 포인터의 크기가 x64 기준으로 16이 된다. 포인터의 크기가 커진 이유는 클래스 구조에 맞게 멤버 함수 포인터를 처리하기 위하여 부가 정보가 포함되기 때문이다.

여기서 함수 포인터의 크기를 소개한 이유는 독자들이 고정 관념을 깨고 더 넓은 시야로 포인터를 비롯하여 프로그래밍 개념을 바라보길 원하기 때문이다.

이 장 첫 부분에서 언급했듯이 함수 포인터를 완벽하게 설명하려면 내용의 수준이 깊어짐과 동

시에 분량도 많아지는 문제가 있다. 또한 C++를 사용하는 데 있어서 꼭 해당 내용을 알아야 할 필요는 없기에 여기서 마무리를 짓는다. 혹시라도 함수 포인터를 더 깊이 있게 공부하고 싶다면 필자의 저서 『Fundamental C++ 프로그래밍 원리』(혜지원, 2015)를 참조하기 바란다.



01 – 반환 타입이 double이고 인자 타입열이 (int, int)인 함수를 가리키는 포인터 pf를 정의하시오.

02 – 01번의 함수 포인터 pf의 타입을 typedef를 통하여 PFunc로 정의하시오.

03 – 다음 프로그램에서 잘못된 부분을 찾아서 수정하시오.

```
int Square(int n)
{
   return n * n;
}

typedef int (*PFUNC)(int, int);

void main()
{
   PFUNC pf = &Square;
   cout << pf(2) << endl;
}
```

04 – 클래스 CTest의 시그니처가 int (double)인 멤버 함수를 가리키는 포인터 pf를 정의하시오.

05 – 멤버 함수 포인터를 통해서 함수를 호출할 경우 반드시 해당 클래스 객체를 기준으로 호출하여야 한다. 이유를 설명하시오.

06 – 05번의 함수 포인터 pf의 타입을 typedef를 통하여 PFunc로 정의하시오.

07 – 다음 프로그램에서 잘못된 부분을 찾고 이유를 설명하시오.

```
class CTestA
{
public:
   void Func()
   {
   }
};

class CTestB
{
public:
   void Func()
   {
   }
};

void main()
{
   void (CTestB::*pf)() = &CTestA::Func;
}
```

{1}- 다음 함수 Sum을 가리키는 함수 포인터 pf를 정의하고 pf를 통해서 1과 2의 합을 출력하는 프로그램을 작성하시오.

```
int Sum(int a, int b)
{
   return a + b;
}
```

{2}- 다음 클래스 CMath의 멤버 함수 Ceil과 Floor를 가리키는 함수 포인터 pf1, pf2를 정의한 후에 함수 포인터를 이용하여 3.5에 대한 Ceil, Floor의 값을 출력하는 프로그램을 작성하시오. Ceil, Floor도 외부 정의하시오.

```
class CMath
{
public:
   CMath(double arg)
   {
      m_Value = arg;
   }

   int Ceil();
   int Floor();

      double m_Value;
};
```

{3}- {1, 3, 5, 9, 7, 4, 2, 6, 8, 0 }에 대하여 qsort를 사용하여 정렬한 결과를 출력하는 프로그램을 작성하시오.