
# 19. 함수 포인터(function pointer)

- 포인터는 Java나 C#같은 고급 프로그래밍 언어에서는 지원하지 않는 개념입니다.
- C/C++은 메모리를 직접 접근하는 등, 저수준의 작업을 할 수 있는데, 이러한 작업을 위해서는 언어 자체가 포인터를 지원하는 것이 반드시 필요합니다.
- 함수 포인터는 동작의 시작, 즉 함수 코드의 시작을 가리키는 포인터입니다.

## 함수 포인터가 필요한 경우

• 사용자로부터 0,1 혹은 2를 입력받아 해당하는 함수를 호출하여 결과를 화면에 출력합니다.

a=5,b=2
Select a Function Number:
0=Exponent(a,b): a**b
1=Multiply(a,b): a*b
2=Divide(a,b): a/b

• 사용자가 0을 입력하면, $5^2$을 출력하며, 1을 입력하면, 5×2=10을 출력하며, 2를 입력하면, 5/2=2를 출력합니다.

```
#include <stdio.h>
#include <math.h>
#include <conio.h>

long Exponent(long a,long b) {
    return pow((double)a,(double)b);
}//Exponent

long Multiply(long a,long b) {
    return a*b;
}//Multiply

long Divide(long a,long b) {
    return a/b;
}//Divide

void main() {
    long a=5,b=2;
    int i;
```

```
    printf("a=5,b=2\n"
           "Select a Function Number:\n"
           "0=Exponent(a,b): a**b\n"
           "1=Multiply(a,b): a*b\n"
           "2=Divide(a,b): a/b\n");
    scanf("%d",&i);
    switch (i) {
        case 0:
            printf("%ld\n",Exponent(a,b));
            break;
        case 1:
            printf("%ld\n",Multiply(a,b));
            break;
        case 2:
            printf("%ld\n",Divide(a,b));
            break;
    }
    getch();
}
```

## 함수 포인터 선언

• 함수 포인터는 포인터 변수의 특수한 종류인데, 함수의 시작 주소를 가지는 포인터 변수입니다.

```
int (*i);
```

• 이 문장은 i가 정수형 포인터가 아니라 정수를 리턴하는 함수형 포인터라는 것을 의미하기 위해 사용합니다.
• 여기에 함수의 원형 정보를 추가해야 합니다. 만약 이 함수 포인터에 정수를 두개 파라미터로 받는 임의의 함수 f(int a,int b); 의 시작주소로 초기화를 원한다면, 다음과 같이 작성해야 합니다.

```
int (*i)(int,int);
```

- getch값을 대입받기 위해서는 다음과 같인 선언해야 합니다.

```
int (*i)();
i=getch;
```

- 이제 i는 메모리에 할당된 getch함수의 시작 주소값을 가집니다.
- getch()를 호출하기 위해서 getch()라고 하듯이 i()라고 사용할 수 있습니다.

```
int (*i)();
int j;
i=getch;
j=i(); // j=getch()와 동일합니다.
```

## 함수 포인터 배열

• 함수 포인터 배열을 선언하기 위해서는 일반 배열이 그런 것처럼 이름 바로 뒤에 []을 붙여주어야 합니다.

```
int  (*i[3])();
```

• 위 문장은 정수를 리턴하는 함수의 주소를 받을 수 있는 크기 3인 함수 포인터 배열을 선언한 것입니다.

```
#include <stdio.h>
#include <math.h>
#include <conio.h>

long Exponent(long a,long b) {
    return pow((double)a,(double)b);
}//Exponent

long Multiply(long a,long b) {
    return a*b;
}//Multiply

long Divide(long a,long b) {
    return a/b;
}//Divide

void main() {
    long (*f[])(long,long)={Exponent,Multiply,Divide};
    long a=5,b=2;
```

```
    int i;

    printf("a=5,b=2\n"
           "Select a Function Number:\n"
           "0=Exponent(a,b): a**b\n"
           "1=Multiply(a,b): a*b\n"
           "2=Divide(a,b): a/b\n");
    scanf("%d",&i);
    printf("%ld\n",f[i](a,b));
}
```

## 오버로드된 함수의 주소

• C++에서 다형성(polymorphism)은 파라미터의 형식이 다른, 같은 이름의 함수를 허용합니다.

```
#include <stdio.h>

void Print(int i) {
    printf("%d\n",i);
}
void Print(char* format,int i) {
    printf(format,i);
}

void main() {
    void (*f)(char*, int);

    f=Print;//f의 정보는 두 번째 Print의 주소가 대입될 것을 알려준다.
    (*f)("hello %d\n",100);
}
```

## 디폴트 파라미터(default parameter)

• 디폴트 파라미터란 함수를 선언할 때, 파라미터의 값을 미리 정해주는 것을 말하는데, 이렇게 정의된 함수를 호출하는 쪽에서, 디폴트 값이 정해진 파라미터를 명시해 주지 않으면, 선언에서 미리 정해진 값이 사용됩니다.

• 예를 들면, void f(int a,int b=10){...} 처럼 정의된 함수의 2번째 파라미터의 디폴트 값은 10입니다.

• f()를 호출할 때, f(1)로 호출할 수 있으며, 2번째 파라미터가 명시되지 않았으므로, f(1,10)의 호출과 같습니다.

• C++에서 함수 포인터에 추가된 사항은 또한 디폴트 파라미터에 관한 것입니다.

• 비록 원래의 함수가 디폴트 파라미터를 가지지 않더라도 함수 포인터에 디폴트를 선언하는 것을 허락합니다.

```
…
void main() {
    void (*f)(char*, int i=0);
    f=Print;//f의 정보는 두 번째 Print의 주소가 대입될 것을 알려준다.
    (*f)("hello %d\n");
}
```

- 결과는 다음과 같습니다.

hello 0

• 표준함수 itoa()는 수(integer)를 ASCII 스트링을 변환하는 표준함수입니다.

```
#include <stdlib.h>
...
char *itoa(int value, char *string, int radix);
...
```

• itoa()의 마지막 파라미터는 변환을 원하는 진수의 베이스(base)입니다. 만약 10진수로 변환을 원한다면, 세 번째 파라미터를 10으로 설정해야 합니다. 이것을 세 번째 값의 디폴트가 10인 함수 포인터로 구현하려고 합니다. 어떻게 해야 할까요?

## 함수 포인터 형의 정의

• 함수 포인터를 빈번하게 사용해야 한다면, typedef를 이용하여 형을 정의하는 것이 바람직합니다.
• 함수 포인터 형의 정의는 좀 특이합니다.
• typedef가 없다면 이것은 함수 포인터 변수의 선언과 동일합니다.

```
typedef long (*Fun)(long,long);
```

• 위에서 정의된 형(type)은 Fun입니다. 이제 Fun은 long을 리턴하고, 파라미터로 2개의 long을 가지는 함수 포인터 변수를 선언하기 위해 형으로 사용할 수 있습니다.

```
Fun f[3];
```

• f[0], f[1], f[2]는 각각 함수 포인터 변수입니다.

```
#include <stdio.h>
#include <math.h>
#include <conio.h>

typedef long (*Fun)(long,long);//Fun is function pointer type

long Exponent(long a,long b) {
    return pow((double)a,(double)b);
}//Exponent

long Multiply(long a,long b) {
    return a*b;
}//Multiply

long Divide(long a,long b) {
    return a/b;
}//Divide

void main() {
```

```
    Fun f[]={Exponent,Multiply,Divide};
    long a=5,b=2;
    int i;

    printf("a=5,b=2\n"
           "Select a Function Number:\n"
           "0=Exponent(a,b): a**b\n"
           "1=Multiply(a,b): a*b\n"
           "2=Divide(a,b): a/b\n");
    scanf("%d",&i);
    printf("%ld\n",f[i](a,b));
}
```

## 진보된 주제: 멤버 함수의 주소

• 객체마다 멤버 함수(member function)가 만들어지는 것은 아님에도 불구하고, 멤버 함수가 파라미터로 받는 this 포인터 때문에 멤버 함수의 주소를 얻는 방법과, 멤버 함수의 주소 변수(member function pointer)를 선언하는 방법은 일반 함수의 것과는 다릅니다.

• int를 리턴하고, void를 파라미터로 받는 CTest의 멤버 함수 포인터를 선언하기 위해서는 다음과 같이 합니다.

```
int (CTest::*fp)();
```

• *CTest::* 표기가 일반 함수의 포인터와 멤버 함수의 포인터를 구분 짓습니다.

• 이 선언이 CTest의 멤버 변수로써의 선언이 아니라는데 유의하세요.

• 위의 선언은 CTest의 멤버 함수의 포인터를 가질 수 있는 일반 변수 - 클래스의 멤버 변수가 아닌 - fp를 선언한 것이지, CTest의 멤버 변수 fp를 선언한 것은 아닙니다.

• 그러므로 아래와 같은 대입문은 모두 타당하지 않습니다(CTest의 객체 t가 만들어졌으며, CTest는 멤버 함수 Get()을 가진다고 가정합니다).

```
...
t.fp=t.Get;//error! fp는 t의 멤버가 아니다.
CTest::fp=t.Get;//error! fp는 CTest의 static 변수가 아니다.
```

• 만약 CTest의 객체 t가 만들어졌다면, t의 멤버 함수 Get()의 시작 주소를 대입하기 위해 다음과 같이 사용하는 것도 가능하지 않습니다.

```
fp=t.Get;
```

• CTest 클래스의 멤버 함수의 주소는 아래와 같이 대입합니다.

```
fp=&CTest::Get;
```

```
#include <iostream>

class CTest {
    int i;
public:
    CTest(int t=0) { i=t; }
    int Get() { return i; }
    void Set(int t) { i=t; }
};

int (CTest::*fp)();//변수 fp는 CTest의 멤버 함수의 주소를 가질 수 있
                   //다.
void main() {
    CTest t(5);

    fp=&CTest::Get;//fp는 CTest 클래스의 Get() 멤버 함수의 시작
                   //소를 가진다.
    std::cout << (t.*fp)() << std::endl;
}//main
```

## C++의 새로운 연산자: .*와 ->*

- C++의 새로운 연산자 .*와 ->*의 기능
- 이 두 연산자는 멤버 함수 포인터가 가리키는 대상을 참조하기 위해 C++에 새롭게 추가된 연산자입니다.
- 일반적으로 점(.)과 화살표(->)는 객체의 멤버를 참조하기 위해 사용되지만, .*와 ->*는 객체에서 객체의 멤버가 아닌 멤버 포인터를 접근하기 위해 사용합니다.

```
#include <iostream>

class CTest {
    int i;
public:
    CTest(int t=0) { i=t; }
    int Get() { return i; }
    void Set(int t) { i=t; }
};
```

```
int (CTest::*fp)();

void main() {
    CTest t(5);
    CTest* tp=&t;

    fp=&CTest::Get;
    std::cout << (tp->*fp)() << std::endl;
                                    //fp는 tp가 가리키는 객체의 멤버가 아니
                                    //지만, ->* 연산자를 사용하여 참조가 가
                                    //능하다.
}//main
```

## 진보된 주제: 멤버 함수 포인터의 응용

• 멤버 함수의 주소가 잘 응용된 곳은 C++ 표준 라이브러리인 iostream에서 찾아 볼 수 있습니다.

• iostream에는 전역 객체인 cout과 cin을 가지고 있는데, 이것은 C의 printf()와 scanf()를 대체합니다.

```
printf("hello₩n");
```

• 위의 printf() 문장에 대응하는 cout의 표현은 다음과 같습니다.

```
std::cout << "hello₩n";
```

- **조작자(manupulator)**라고 불리는 특별한 endl() 함수의 기능을 이용하여 줄을 바꾸는 기능을 포함한 문장을 다음과 같이 작성할 수 있습니다.

```
std::cout << "hello" << std::endl;
```

- endl()은 함수입니다. 그러므로 아래의 표현식은 함수의 주소를 의미합니다.

```
endl
```

- cout에는 함수의 주소를 받는 특별하게 오버로드된 << 연산자 함수가 있습니다.

• cout이란 전역 객체와 조작자 endl을 사용한 - 를 모방한 아래 예제를 참고하세요.

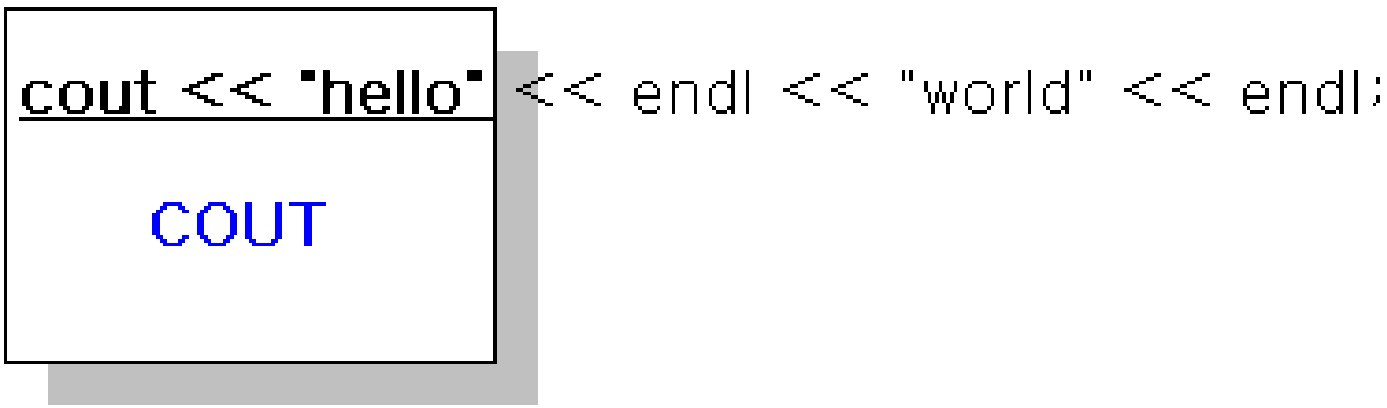
```
#include <stdio.h>

class ostream;
typedef void (*Manipulator)();

void endl() {
     printf("\n");
}//endl

class ostream {
public:
     ostream& operator<<(char* s);
     ostream& operator<<(Manipulator m);
};//class ostream

ostream& ostream::operator<<(char* s) {
     printf("%s",s);
```

```
    return *this;
}//ostream::operator<<

ostream& ostream::operator<<(Manipulator m) {
    (*m)();
    return *this;
}//ostream::operator<<

ostream cout;

void main() {
    cout << "hello" << endl << "world" << endl;
}//main
```

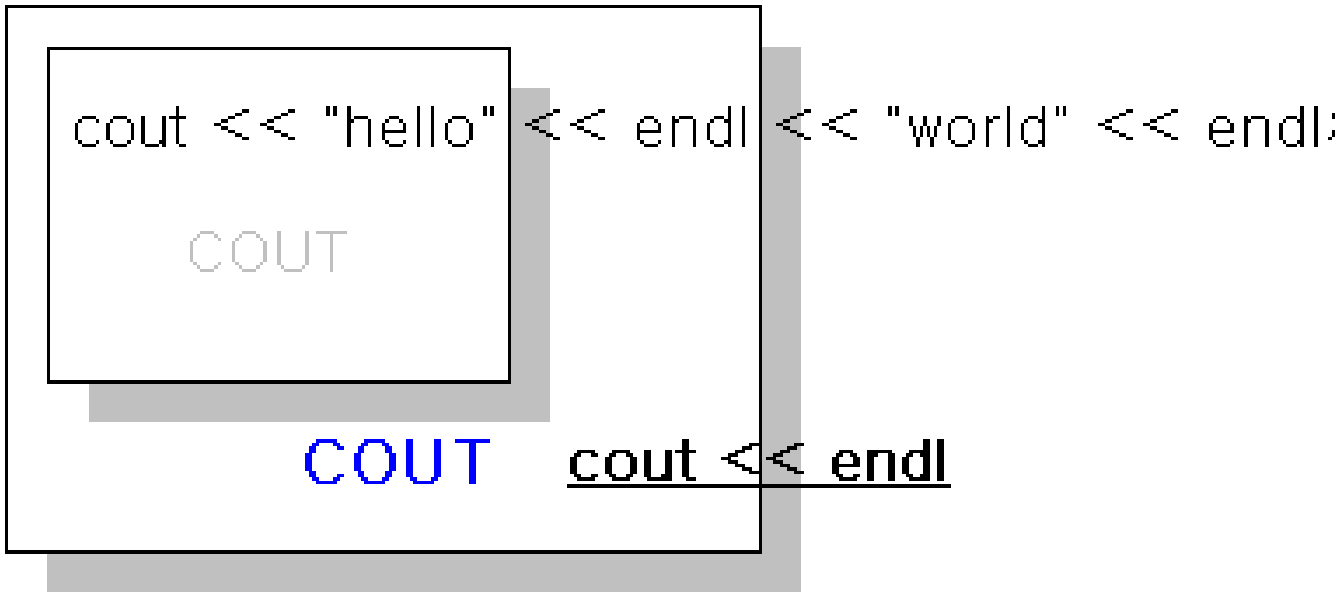
- 출력 결과는 다음과 같습니다.

```
hello
world
```

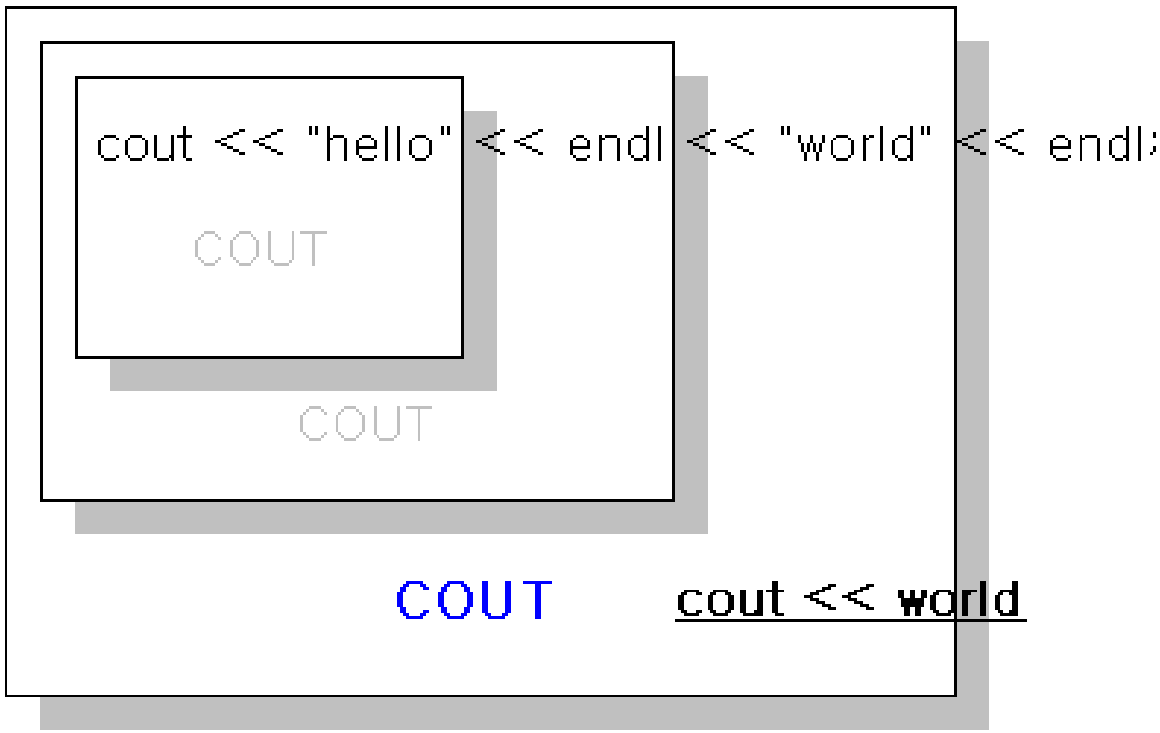
- cout을 사용한 문장이 처리되는 과정은 다음과 같습니다.



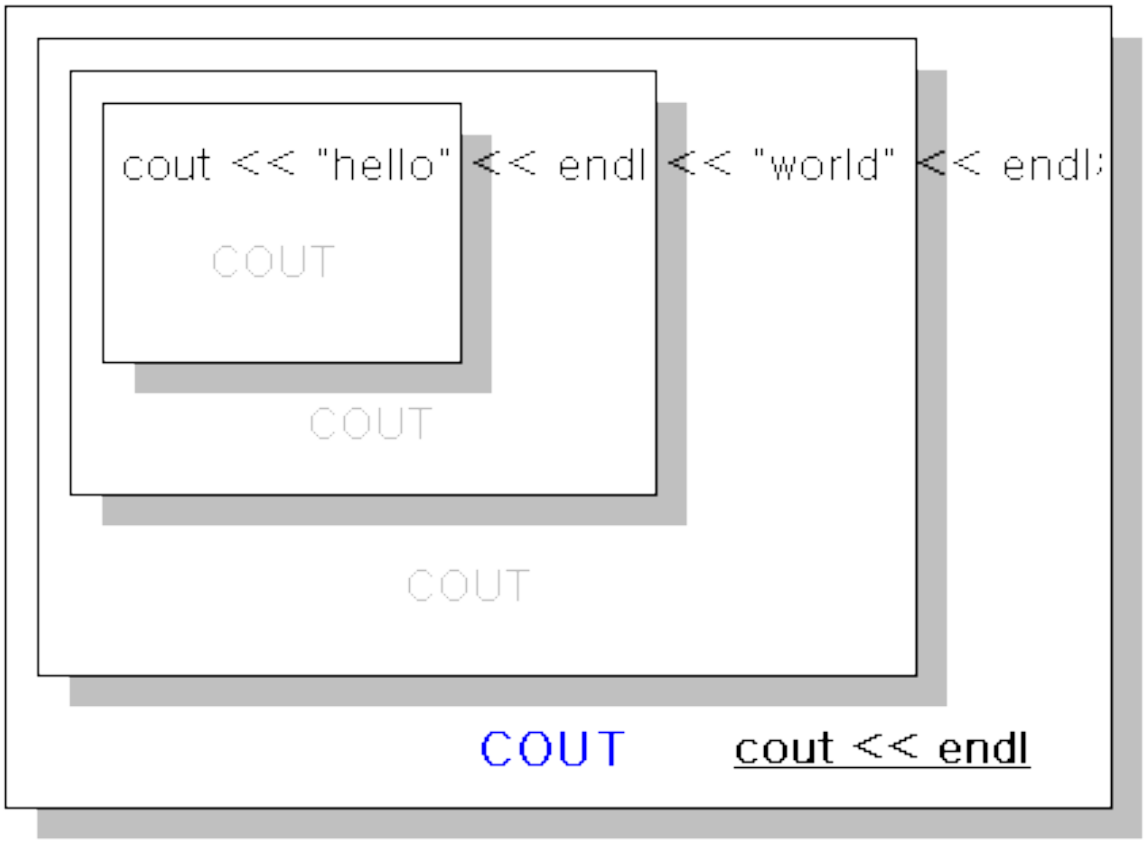

cout << "hello": 이 문장은 ostream& operator<<(char* s);를 호출합니다. ostream의 레퍼런스가 리턴되므로, cout << "hello"는 cout자체입니다.




cout << endl; 이 문장은 ostream& operator<<(Manipulator m);를 호출합니다. 역시 ostream의 레퍼런스가 리턴되므로, cout << endl 역시 cout자체입니다.




cout << "world": 이 문장은 ostream& operator<<(char* s);를 호출합니다. ostream의 레퍼런스가 리턴되므로, cout << "hello"는 cout자체입니다.




cout << endl; 이 문장은 ostream& operator<<(Manipulator m);를 호출합니다. 역시 ostream의 레퍼런스가 리턴되므로, cout << endl 역시 cout자체입니다.

## 실습문제

1. STL(standard template library)의 어떤 부분은 반드시 함수 포인터로 구현해야 하는 상황이 발생합니다. 어떤 부분이 그러하며, 왜 그런가요?

2. itoa(int value, char *string, int radix)의 세 번째 파라미터 radix가 디폴트 값 10을 가지도록 하는 함수 포인터를 선언하고 사용하는 예를 작성하세요.

@