Functions

프로그래밍 입문(2)

Topics

- Function Basics
- **More about Functions**
- **Default Parameter**
- **Call by Value, Address, Reference**
- **Recursion**

Default Parameter

```
void print(int x, int y = 5, int z = 10) {
    cout << "x:" << x << endl;
    cout << "y:" << y << endl;
    cout << "z:" << z << endl;
}
```

```
print(1, 2);  ⟶  x: 1
                 y: 2
                 z: 10

print(1);     ⟶  x: 1
                 y: 5
                 z: 10
```

- 각각의 매개변수별로 기본값(default)을 정의할 수 있음.
- 함수 호출시 그 변수의 값(인자)이 주어지지 않는다면, 기본값이 사용됨.
- 기본값은 가장 오른쪽 매개변수부터 지정이 가능하며, 중간에 생략할 수 없음.

Default Parameter

- 기본값을 중간에 생략하거나, 왼쪽부터 정의하는 형태가 안되는 이유는 함수의 호출 때문.

- 함수 호출시 왼쪽부터 매개변수와 인자를 매칭.

- 중간에 인자를 건너뛰는 형태의 함수 호출은 허용되지 않음.

중간에 생략 → X

```
void print(int x = 2, int y, int z = 10)

print(,3,);  Error!
```

왼쪽부터 정의 → X

```
void print(int x = 2, int y, int z)

print(,2,3);  Error!
```

Default & Overloading

- 기본값을 지정한다는 것은 인자의 생략을 고려한다는 것.
- 인자가 생략된 형태가 다른 오버로드된 함수의 매개변수 형태와 일치한다면?
  - 함수 호출시 일치하는 정의된 함수가 하나 이상이 되므로 에러가 발생함.

```
void print(int x, int y, int z = 5) {
    cout << "x:" << x << endl;
    cout << "y:" << y << endl;
    cout << "z:" << z << endl;
}

void print(int x, int y) {
    cout << "x:" << x << endl;
    cout << "y:" << y << endl;
}

print(1, 2);
```

Default & Prototype

- 프로토타입에서도 기본값을 선언할 수 있음.

```
void print(int x, int y = 5, int z = 10);
```

- 이 경우 함수의 정의에서는 매개변수에 기본값을 지정할 수 없음.

- 동일한 기본값 지정 (z=10)이나, 프로토타입에 기본값이 선언되지 않은 변수의 기본값 지정 (x=2) 등도 허용되지 않음.

```
void print(int x, int y, int z = 5) {
    cout << "x:" << x << endl;
    cout << "y:" << y << endl;
    cout << "z:" << z << endl;
}
```

```
: error:
      redefinition of default argument
void print(int x, int y, int z = 5) {
                                 ^   ~
```

Call by Value

- 기본적으로 함수 호출시 인자는 함수에게 값만 전달됨.
- 함수호출 후의 진행.
- 인자로 넘어온 값들을 메모리에 저장.
- 실행을 함수로 넘김.
- 함수에서는 저장된 인자들의 값을 읽어들임.
- 함수 내부 코드 실행.

```
int increase(int x, int y) {
    x += y;
    return x;
}
```

```
int x = 10;
increase(x, 5);
```

호출 후 x의 값은?

```
x = increase(x, 5);
```

반환된 값을 다시 대입하는 경우,
호출 후 x의 값은?

Call by Address

- 배열 등을 인자로 넘기는 경우.
- 배열 변수는 실제 배열이 시작되는 메모리의 주소값을 가짐.
- 이 메모리 주소값이 복사되어 Call by Value와 똑같이 동작하게 됨.
- 실제 함수 내부에서는 이 주소값을 이용해 동일한 메모리에 접근 가능.
- 실제 데이터를 전부 복사하지 않아 성능이 좋음.

포인터형 매개변수
→ 주소값을 가짐

```
void increase(int *a, int y) {
    a[0] += y;
}

int a[1] = { x };
increase(a, 5);
```

호출 후 함수 내부에서 a는 인자로 넘긴 배열 a와 동일한 주소값을 가짐.

a[0]로 접근하는 메모리 위치는 함수 안이나 밖 모두 동일하므로, **함수에서 그 위치의 값을 바꾸면 함수 밖에도 반영됨**.

Call by Reference

- 참조(reference)는 C++에서 새로 도입된 개념.
- 참조형 변수는 &를 사용하여 표시할 수 있음.
- 함수의 매개변수로 참조형 변수가 사용되면 인자로 지정된 변수의 참조가 실제로 함수 내부에서 사용됨.
- 특정 변수에 대한 별칭이 하나 더 생성되는 것으로 생각하면 됨.
- 실제 데이터를 전부 복사하지 않아 성능이 좋음.

```
void ref_increase(int &x, int y) {
    x += y;
}
```

값만 복사된 것이 아닌, 실제 동일한 위치를 가리킴.

```
ref_increase(x, 5);
```

함수 호출후 x의 값이 변하게 됨.

Recursion

```
int fact(int n) {
    if(n == 1 || n == 2)
        return n;
    else
        return n * fact(n-1);
}
```

- 재귀(Recursion)는 자신을 정의할 때 자기 자신을 재참조 하는 형태를 의미함.
  - *Factorial of n = 1 x 2 x 3 x … x n ⇒ (Factorial of n - 1) x n*
- 함수에서는 **재귀 호출(recursive call)**의 형태로 나타남.
- 수학적 정의나 알고리즘 등이 자연스럽게 재귀 형태를 띄는 경우가 많으므로, 이를 쉽게 표현할 수 있음.

Recursion

```
int fact(int n) {
    if(n == 1 || n == 2)        terminating case
        return n;
    else
        return n * fact(n-1);   recursive call
}
```

- 재귀 함수를 구현할 때는 반드시 다음의 두 가지 사항을 주의해야함.

- 최종적으로 더 이상 자신을 호출하지 않고 종료하는 경우(terminating case)의 코드를 넣어주어야 함.

- 재귀 호출을 할 때는 반드시 인자를 변화시켜 종료하는 경우에 도달할 수 있게 해야함.

  - e.g.) fact(int n)에서 fact(n)을 다시 호출해서는 안 됨.

Recursion

- 재귀 형태(Recursive)로 구현된 함수는 반복 형태(Iterative)로도 구현할 수 있음.

- 함수 호출에 드는 비용(overhead)가 줄어들어 반복 형태가 더 성능이 좋은 경우가 많음.

- 구현 자체는 재귀 형태가 더 직관적인 경우가 많음.

```
int fact(int n) {
    if(n == 1 || n == 2)
        return n;
    else
        return n * fact(n-1);
}
```

```
int iter_fact(int n) {
    int fact = 1;
    for(int i=1; i<=n; i++)
        fact *= i;
    return fact;
}
```

Fibonacci Number

- n번째 Fibonacci number Fib(n)
  - Fib(n) = Fib(n-1) + Fib(n-2)
- 재귀 형태로 쉽게 구현 가능함.
- 반복 형태로도 2개의 임시변수를 도입하여 구현 가능.
- 재귀 형태는 심각한 성능저하가 있음.
  - fibonacci(4)를 호출하면, fibonacci(3) + fibonacci(2)를 반환함.
  - 이 때 fibonacci(3)의 호출은 내부적으로 다시 fibonacci(2)를 호출하므로 중복이 심하게 일어남.

```c
int fibonacci(int n) {
    if(n == 0 || n == 1)
        return n;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}

int iter_fib(int n) {
    int prevPrevNum, prevNum = 0, currNum = 1;
    for(int i=1; i<n; i++) {
        prevPrevNum = prevNum;
        prevNum = currNum;
        currNum = prevPrevNum + prevNum;
    }
    return currNum;
}
```

재귀함수 고려사항

- 재귀 함수를 구현할 때는 반복형태가 쉽게 가능하지 않은지 고민해 볼 필요가 있음.

- 새롭게 함수를 호출하는데 드는 오버헤드가 있다는 점을 잊지 말아야 함.

- 복잡한 알고리즘 구현시 간결하고 이해하기 쉬운 코드를 작성하기 위해 재귀함수 사용이 불가피할 경우가 있음.

  - e.g.) Quick/Merge Sort, Tree 탐색 알고리즘 등.

Summary

- Default Parameter
- Call by Value, Address, Reference
- Recursion