객체지향 프로그래밍 노트
전달 인자의 수가 가변적인 경우

강대기

2007년 11월 3일

본 고에서는 전달 인자의 수가 가변적인 경우에 대해 논해보고자 한다. 이 부분에 대해 본 교재인 "C++ 기초 플러스 5판"에서는 354 쪽과 355 쪽에서 아주 간단하게 논하고 있다.

우리가 C에서 가장 먼저 배우는 printf 함수를 고려해 보자. 일단 printf는 라이브러리 함수로 C의 키워드는 아니다.

그런데, printf 함수는 몇 개의 인자들이든 취할 수 있는 거처럼 보인다. 예를 들어 다음이 모두 제대로 돌아가는 C 명령인 것이다.

```
printf("Hello World\n"); // 1
printf("%d\n", x); // 2
printf("%d, %f\n", x,y); // 3
printf("%d, %f, %c\n", x,y,z); // 4
printf("%s, %d, %f, %c\n", "Hello world", x,y,z); // 5
printf("%s, %d, %d, %d, %d, %d, %d, %d\n", "Hello world", 1,2,3,4,5,6,7); // 6
```

이것은 어떻게 가능한 것일까? 가만히 위의 인자들을 보면, 맨 처음 인자만 가지고 뒤에 몇 개의 인자가 오게 될지 판별할 수 있다는 사실을 알 수 있다. 2번째 경우는 %d 가 있으므로 뒤에 한 개가 올 것이고, 3 번의 경우는 %d 와 %f 가 있으므로 2 개가 올 것이다.

몇 개의 인자들이든 취할 수 있다는 말은 정확하게 고치면, 실은 전달 인자를 저장하는 메카니즘이 허용하는 한 (보통 스택으로 구현된다) 몇 개의 인자든지 가능하다고 말할 수 있다. 기본적으로 Java 언어를 포함한 대부분의 언어들은 전달 인자들을 스택에 넣는 데, C/C++의 경우는 전달 인자들을 맨 뒤의 것부터 거꾸로 넣는다. 중요한 점은 이렇게 함으로써 몇 개의 인자를 스택에 넣건 맨 첫번째 인자는 접근이 가능하다는 것이다.

예를 들면 다음의 printf 문을 보자.

```
printf("%d, %s\n", 100, str);
```

이 printf 문을 호출하면 스택은 다음 그림 1과 같이 된다.

그림 1처럼 몇 개의 인자를 보내서 스택에 쌓 건간에, 첫번째 인자의 주소는 printf 함수 내에서 찾아갈 수 있다. 즉 함수 내의 로컬 변수가 있는 위치에서 내려가서 복귀 정보를 지나 첫번째 인자를 찾을 수 있는 것이다. 이 첫번째

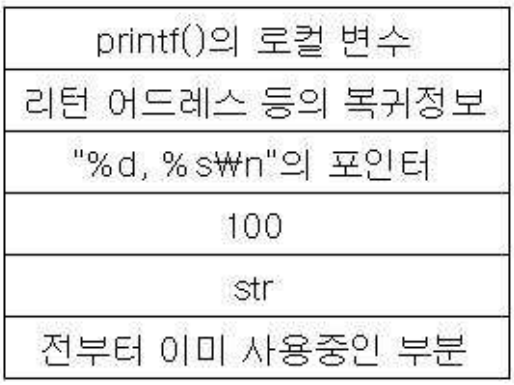

그림 1: 가변 인자들을 가진 함수를 호출했을 때의 스택의 구조

인자를 해석하면 스택의 아래 부분에 몇 개의 어떤 인자들이 쌓여 있는지를 알 수 있는 것이다.

즉 첫번째 인자가 매우 중요한 셈이다. 첫번째 인자에서 뒤에 오는 인자들의 개수와 타입들에 대한 정보를 줄 수 있어야 한다.

이러한 점들을 고려해서 우선 가변 인자이지만 인자들의 타입을 정수로 제한하고 첫번째 인자에서 인자들의 개수를 지정하는 간단한 예부터 시작해 보자.

```
#include <cstdio>

void printInt(int n,...)
{
  switch (n)
  {
    case 1: fprintf(stdout,"%d\n",*(&n+1)); break;
    case 2: fprintf(stdout,"%d,%d\n",*(&n+1),*(&n+2)); break;
    case 3: fprintf(stdout,"%d,%d,%d\n",*(&n+1),*(&n+2),*(&n+3)); break;
    default: fprintf(stderr,"PANIC: wrong number of arguments :%d\n",n);
  }
}

int main()
{
  printInt(1,10);
  printInt(2,10,20);
  printInt(3,10,20,30);
  int a=100, b=200, c=300;
  printInt(1,a);
  printInt(2,a,b);
  printInt(3,a,b,c);
```

```
  return 0;
}
```

이 프로그램을 실행하면 다음과 같은 결과가 출력된다.

```
10
10,20
10,20,30
100
100,200
100,200,300
```

이러한 프로그램은 &n이 int 형 주소이므로 다음과 같이 포인터로 표현할 수도 있다.

```
int* ap = &n+1;
```

다음은 이러한 포인터와 for 루프로 앞의 프로그램을 표현한 것이다.

```
#include <cstdio>

void printInt(int n,...)
{
  int* ap = &n+1;
  for (int i=0;i<n;i++) fprintf(stdout,"%d,",ap[i]);
  fprintf(stdout,"\n");
}

int main()
{
  int a=100, b=200, c=300;
  printInt(1,a);
  printInt(2,10,20);
  printInt(3,a,b,c);
  return 0;
}
```

이 프로그램을 실행하면 다음과 같은 결과가 출력된다.

```
100,
10,20,
100,200,300,
```

결국 근본적으로 이러한 식으로 가변 인자를 다루고 있는 셈이다. 그런데, 지금까지의 예는 정수 인자인 경우만 다루고 있다. 인자의 크기를 확인하고 주소를 계산하는 것은 그다지 어렵지는 않아 보이지만 포인터가 관련되어 있으므로 다소 귀찮다. 또한 정수 인자들만 있는 건 아니므로 인자들의 타입들도 고

려해야 하며, 모든 타입들이 얼마만큼의 메모리를 차지하는가는 시스템에 따라 다르다.

이러한 점들을 고려해서 va_list, va_start, va_arg, va_end 등의 매크로를 사용할 수 있다. 이 매크로들을 사용하려면 stdarg.h라는 ANSI C 표준 헤더 파일을 포함시켜야 한다.

```
#include <stdio.h>
#include <stdarg.h>

int add_up (int count, ...)
{
  va_list ap;
  va_start (ap, count); // 인수 목록 초기화
  int sum = 0;
  for (int i = 0;i < count;i++)
    sum += va_arg (ap, int);  // 다음 인수
  va_end (ap);
  return sum;
}

int main (void)
{
  // 16 출력
  printf ("%d\n", add_up (3, 5, 5, 6));
  // 55 출력
  printf ("%d\n", add_up (10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10));
  return 0;
}
```

va_list ap; 명령은 스택에 저장되어 있는 인수들을 지정해서 읽어들이기 위한 포인터를 선언하는 매크로이다. va_start(ap,첫번째 인자) 매크로는 가변 인자들을 읽기 위해 ap 포인터 변수가 첫 번째 가변 인자를 가리키도록 초기화한다.

va_arg(ap,인자 타입) 매크로는 가변 인자를 실제로 읽는 명령이다. va_start가 ap를 첫 번째 가변 인자의 주소로 지정해 주므로 기본적으로 ap로 그 인자를 가리켜서 읽게 하는 것이다. 일단 읽은 후에는 ap의 위치를 자동으로 현재 읽은 인자의 바로 다음 주소로 설정해서, 다음 인자가 있다면 읽을 수 있게 한다. 그런데, ap 번지에 있는 인자가 어떤 타입인지를 지정해야 va_arg 매크로가 인자 값을 제대로 읽을 수 있으므로 va_arg의 두 번째 인자로 읽고자 하는 값의 타입을 지정 한다. 예를 들어 ap 위치에 있는 int 값을 읽고자 한다면 va_arg(ap, int)를 호출하고 double 값을 읽고자 한다면 va_arg(ap, double)이라고 호출하면 된다. va_arg의 두 번째 인수는 내부적으로 sizeof 연산자와 형 강제 즉 타이프캐스트(typecast) 연산자로 전달된다.

va_end(ap) 매크로는 기본적으로 va_start와 짝을 이루기 위해 만든 매크로인데, 겉보기에는 마치 가변 인자들을 다 읽은 후 뒷 정리를 하는 것처럼 보이지만, 대부분의 시스템에서는 실제로는 아무 짓도 하지 않으며 따라서 기본

적으로는 없어도 된다. 그러나 이 명령은 짝을 이루기 위한 문제와 호환성 문제 때문에 그대로 넣어주는 게 좋다. 다른 플랫폼이나 장래의 새로운 컴퓨터나 운영 체제의 환경에서는 va_end가 중요한 역할을 할 수도 있기 때문이다.

이제 어느 정도 예제 프로그램들을 통해 이론을 이해했을 것이나, 더 나아가 단순히 정수형인 경우 말고, double 과 char도 받을 수 있는 예제를 보도록 하겠다.

```
#include <cstdio>
#include <cstdarg>
#include <cassert> // assert.h

void myPrint(const char *fmt, ...)
{
  va_list ap;
  va_start(ap,fmt);
  for (int i=0;fmt[i];i++)
  {
    switch (fmt[i])
    {
      case 'i': // int
        printf("%d",va_arg(ap,int));
        break;
      case 'd': // double
        printf("%f",va_arg(ap,double));
        break;
      case 'c': // char
        printf("%c",va_arg(ap,char));
        break;
      case 'f': // float
        printf("%f",va_arg(ap,float));
        break;
      case 's': // char*
        printf("%s",va_arg(ap,char*));
        break;
      default: // Error
        assert(0); // 심각한 에러이므로 바로 종료
    }
  }
  va_end(ap);
}

void main()
{
  myPrint("sisis","The results are ",10," and ",20,".\n");
  myPrint("icicic",10,'+',20,'=',30,'\n');
  myPrint("kiss","kiss:", 100,100); // 일부러 에러 발생
```

```
}
```

이 프로그램은 처음에 제대로 접근할 수 있는 첫번째 인자로 C 스타일 스트링을 받고 그 내용을 한 문자씩 분석하면서 다음 인자의 타입을 알아내어 출력을 하고 있다.

한가지 언급하고 싶은 것은, C 표준에 따르면 가변 인자를 처리하는 함수에서 float 인자, 즉 `va_arg(argp, float)`을 쓰면, 이른바 "default argument promotion"이 적용된다. 즉, float 타입의 인자들은 항상 double로 변환되며, char나 short int의 경우 항상 int로 변환된다. 따라서, 엄밀히 말하면 `va_arg(argp, float)`은 잘못된 코드이며, 대신 `va_arg(argp, double)`을 써야 한다. 같은 이유로, 실전에선 char, short, int를 받기 위해서는 `va_arg(argp, int)`를 써야 한다. 그러나, 여기서는 이해를 돕기 위한 예제 차원에서 넣었다.

위의 프로그램에서 myPrint를 세 번 호출하는 데, 마지막에는 일부러 지정되지 않은 문자인 'k'를 집어넣었다. 이런 경우, `assert(0);`으로 바로 종료하게 만들었다. 이 프로그램을 실행하면 다음과 같은 결과가 나온다.

```
The results are 10 and 20.
10+20=30
Assertion failed: 0, file c:\users\ (생략) .cpp, line 29
```

여기서 한 가지 문제점을 알 수 있는 데, 인자들에 대한 형 검사, 즉 타입 체킹에 대해 컴파일러에서는 아무런 도움도 주지 않고, 코딩하는 프로그래머가 알아서 챙겨야 한다는 것이다. 이건 C에서 가변 인자를 사용하는 경우, 피할 수 없는 약점 중 하나이며, printf 같이 우리가 자연스럽게 많이 사용하는 함수에 대해서도 마찬가지다.

```
#include <cstdio>
void main()
{
  printf("%d, %d\n",1,2);
  printf("%d, %d, %d, %d, %d\n",1,2); // 인자가 더 적다.
  printf("%d, %d, %d\n",1,2,3,4,5); // 인자가 더 많다.
  printf("%d, %d\n",1,3.14); // 두번째 인자의 타입이 틀렸다.
  printf("%f, %f\n",1,2); // 모든 인자들의 타입이 틀렸다.
  printf("%s\n",1); // 인자의 타입이 심각하게 틀렸다.
}
```

이 프로그램을 실행하면 필자의 시스템에선 앞의 다섯 개의 printf만 실행되고, 마지막 printf 문에 대해서는 에러가 발생하고 종료하였다. 인자의 개수가 더 많거나 적은 경우에 대해 어떻게 행동하는가나, 인자의 타입이 틀린 경우에 대해서는 컴파일러나 운영체제에 따라 다르며, 한마디로 정의되어 있지 않아 예측불허이다.

```
1, 2
1, 2, 0, 0, 2147340288
1, 2, 3
1, 1374389535
0.000000, 0.000000
// 여기서 에러 발생하고 종료
```

앞의 그림 1에서 스택 프레임을 소개했다. 이 스택 프레임은, 실제 스택 프레임의 내용을 그대로 반영한다기 보다는, 다소 개념적인 모습이다.

이제 마지막으로 스택 프레임의 내용을 실제로 변경하는 프로그램을 소개해 보겠다.

```
#include <iostream>
#include <cstdlib>

using namespace std;

void fun(void);
int function(int, int);

int main()
{
  int a =100, b = 200;
  int c = function(a, b);
  cout << "정상적인 종료!" << endl;
  cout << "[ " << a << " + " << b << " = " << c << " ] " << endl;
  cout << "하지만 여기까지 오지는 않는다." << endl;
  return 0;
}

void fun()
{
  cout << "하하하.. 어느새 내가 실행되지롱!" << endl;
  exit(0);
}

int function(int x, int y)
{
  void *fun_array[2];
  void **xxx = fun_array + 4;
  *(xxx) = fun; // 리턴 주소를 fun 이라는 함수의 주소로 바꿈
  cout << "function 이라는 함수 실행!" << endl;
  return x+y;
}
```

function 이라는 함수 내에서 `fun_array` 라는 void 포인터 두개의 배열을 선언한다. 이 `fun_array`는 function 함수 내의 최초의 지역 변수이다. 이 지역 변수는 자동 변수이며, C 언어 표준에선 정해져 있지 않지만, 스택에 들어간다.

이제 또 다른 로컬 변수인 void 포인터의 포인터인 xxx 를 정의하고, 최초의 로컬 변수인 `fun_array` 에 4를 더한 값으로 초기화한다. xxx에 들이었는 이 값은 윈도우즈의 스택 프레임을 구성할 때 관례적으로 스택에 들어가는 ebp 레지스터를 건너 뛰어, 리턴 어드레스를 바로 가리킨다. `*(xxx) = fun;` 명령은 이 리턴 어드레스를 fun 이라는 함수의 주소로 바꾼다. 따라서, function에서 일을

전부 수행하고 나서, 원래의 주소로 돌아가는 대신, fun 이라는 함수가 있는 주소로 간다. fun에서는 자신의 일을 수행하고 나서, 스택 프레임이 비워져 있으므로, 그냥 바로 프로그램의 실행을 종료하는 것(`exit(0)`)이다.

프로그램의 실행 결과는 다음과 같다.

```
function 이라는 함수 실행!
하하하.. 어느새 내가 실행되지롱!
```