시스템 프로그래밍 기초

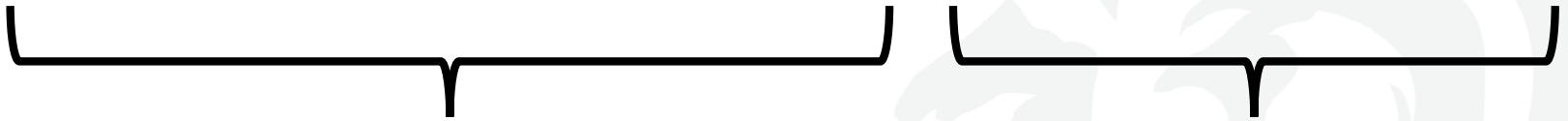
Introduction to System Programming

ICT융합학부 조용우

5. Functions

함수 정의

- *type function_name (parameter list) { declarations statements }*



function head

function body

함수 정의의 예

```
int factorial(int n)    /* header */
{                       /* body starts here */
   int    i, product = 1;
   for (i = 2; i <= n; ++i)
      product *= i;
   return product;
}
```

함수 정의의 예

```
void wrt_address(void)
{
   printf("%s\n%s\n%s\n%s\n%s\n\n",
      "      **********************",
      "      **   SANTA CLAUS    *",
      "      **   NORTH POLE     *",
      "      **   EARTH          *",
      "      **********************");
}
```

```
void wrt_address(void)
```

- 첫번째 void: 이 함수는 return값이 없음
- 두번째 void: 이 함수는 인수를 받지 않음
- 함수를 호출하는 방법

```
wrt_address()
```

- (예)3번 호출

```
for (i = 0; i < 3 ; ++i)
    wrt_address();
```

함수 정의의 예

```
void nothing(void) { }      /* this function does nothing */

double twice(double x)
{
    return (2.0 * x);
}

int all_add(int a, int b)
{
    int c;
    ...
    return (a + b +c);
}
```

함수형이 정의되지 않은 경우

```
all_add(int a, int b)
{
    ...
```

- 함수형을 구체적으로 밝히지 않은 경우
  - → 기본적으로 int형으로 간주
  - → 함수형을 정확하게 명시하는 것이 바람직함 (exercise 5 참조)

지역변수와 전역변수

```
#include <stdio.h>
int    a = 33;      /* a is external and initialized to 33 */

int main(void)
{
    int       b = 77;          /* b is local to main() */
    printf("a = %d\n", a);     /* a is global to main() */
    printf("b = %d\n", b);
    return 0;
}
```

return 문

- *return_statement* ::= return; | return *expression* ;
- (예)

```
return;

return ++a;

return (a * b);
```

- return 문에 도달하면 해당 함수 종료

함수형에 일치하도록 변형되어 리턴

```
float f(char a, char b, char c)
{
    int i;
    ...
    return i; /* the value returned will be converted to a float
*/
}
```

return문은 없거나 (괄호에서 종료) , 여러 개 있을 수도 있음

```
double absolute_vale(double x)
{
    if (x >= 0.0)
        return x;
    else
        return -x;
}
```

받은 return값을 사용하지 않아도 됨

```
while (...) {
   getchar();     /* get a char, but do nothing with it */
   c = getchar();/* c will be processed */
   ...
}
```

함수 원형(prototype)

- *type function_name (parameter list);*
- 함수원형: 컴파일러에게 인자 type, 인자 수, return 값의 형을 알림

(예)

```
double   sqrt(double);
```

→ double형 인자 한 개 갖음, 리턴되는 값은 double형

- 매개변수 형 리스트(parameter type list)는 콤마로 분리
- 식별자는 원형에 영향 없음 (생략해도 괜찮음)

(예)

```
void f(char c, int i);

void f(char, int);
```

거듭 제곱표를 생성하는 예제

```
#include <stdio.h>
#define   N   7

long    power(int, int);
void    prn_heading(void);
void    prn_tbl_of_powers(int);

int main(void)
{
    prn_heading();
    prn_tbl_of_powers(N);
    return 0;
}

void prn_heading(void)
{
    printf("\n::::: A TABLE OF POWERS :::::\n\n");
}
```

거듭 제곱표를 생성하는 예제

```
void prn_tbl_of_powers(int n)
{
    int   i, j;
    for(i = 1; i <= n; ++i) {
        for(j = 1; j <= n; ++j)
            if(j == 1)
                printf("%1d", power(i, j));
            else
                printf("%9ld", power(i, j));
        putchar('\n');
    }
}

long power(int m, int n)
{
    int   i;
    long  product = 1;
    for (i = 1 ; i <= n; ++i)
        product *= m;
    return product;
}
```

거듭 제곱표를 생성하는 예제

```
::::: A TABLE OF POWERS :::::

1        1        1         1         1         1         1
2        4        8        16        32        64       128
3        9       27        81       243       729      2187
4       16       64       256      1024      4096     16384
5       25      125       625      3125     15625     78125
6       36      216      1296      7776     46656    279936
7       49      343      2401     16807    117649    823543
```

컴파일러 관점에서의 함수선언

- 만약 함수선언이나 정의 또는 함수원형의 선언 전에 함수 f(x)가 호출되었다면, 컴파일러는 기본적으로 다음과 같은 형태로 함수가 선언되었다고 가정

```
int f();          /* default declaration */

int f(x)          /* traditional C style */
double x;
{
    ...

int f(double x)   /* ANSI C style */
{
    ...
```

함수정의 순서의 다른 형태

```
#include <stdio.h>
#define   N   7

void prn_heading(void)
{
     ...
}

long power(int m, int n)
{
     ...
}

void prn_tbl_of_powers(int n)
{
     ...
     printf("%ld", power(i, j));
     ...
}

int main(void)
{
    prn_heading();
    prn_tbl_of_powers(N);
    return 0;
}
```

함수정의 순서의 다른 형태

- 하향식 함수 작성
  - → 함수의 정의(definition)은 원형(prototype)의 역할도 함께 수행
  - → 각 함수들에서 호출할 함수가 있는 경우, 해당 피호출 함수의 정의를 먼저 작성
- 예제
  - → **`main()`**를 맨 뒤에 작성
  - → **`main()`**에서 호출할 **`prn_heading()`**과 **`prn_tbl_of_powers()`**의 정의를 먼저 작성
  - → **`prn_tbl_of_powers()`**에서 호출할 **`power()`**의 정의를 먼저 작성

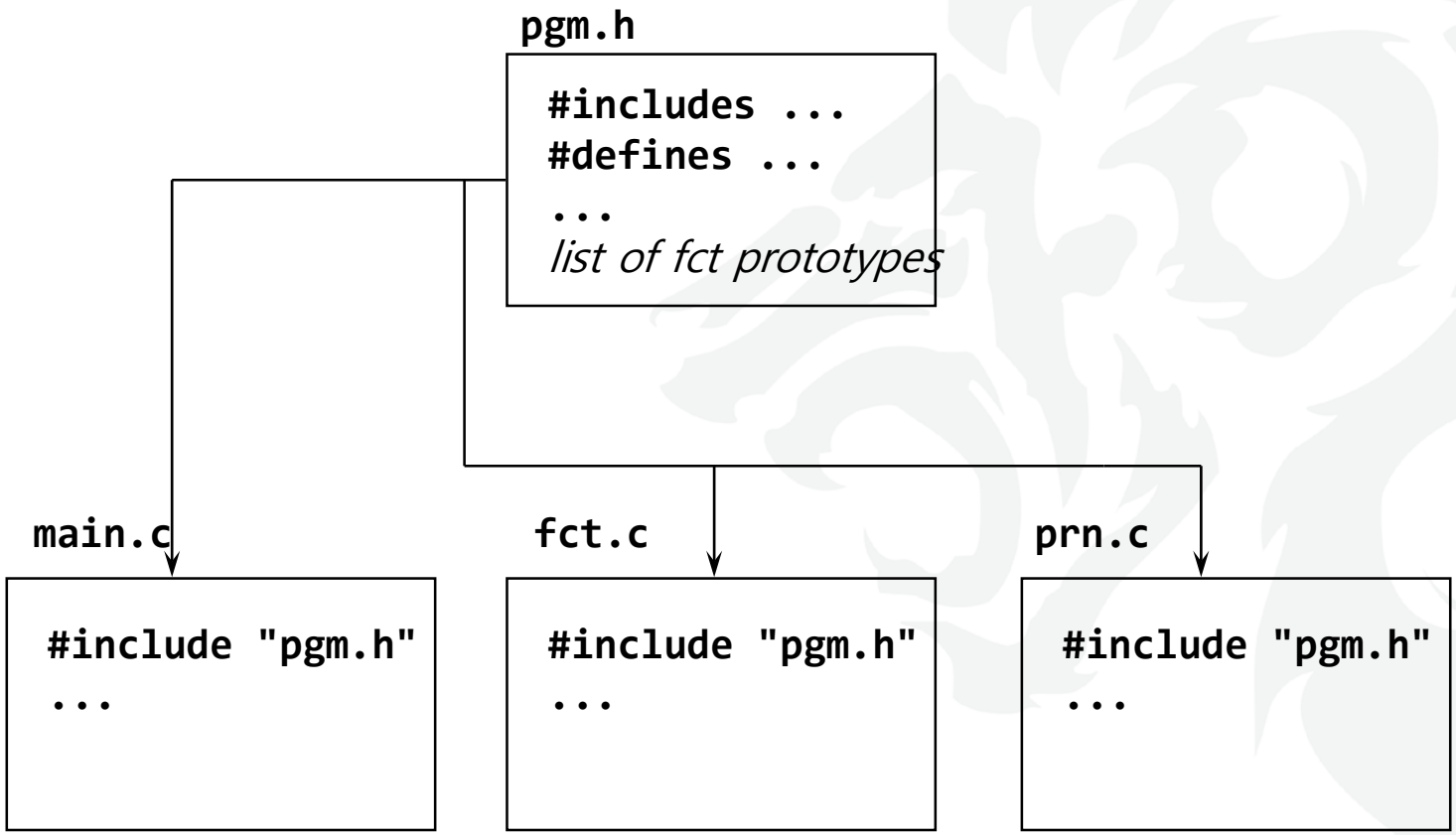
값에 의한 호출

- 값에 의한 호출(call by value)
- 인자는 지역적(local)으로 사용, 호출한 환경에 저장된 변수 값 변경 없음
- 6.3 Call-by-Reference 참조

5.7 Function Invocation and Call-by-Value

```
#include <stdio.h>
int compute_sum(int n);
int main(void)
{
    int  n = 3, sum;
    printf("%d\n", n);       /* 3 is printed */
    sum = compute_sum(n);
    printf("%d\n", n);       /* 3 is printed */
    printf("%d\n", sum);     /* 6 is printed */
    return 0;
}
int compute_sum(int n)       /* sum the integers from 1 to n */
{
    int sum = 0;
    for ( ; n > 0; --n)      /* stored value of n is changed */
        sum += n;
    return sum;
}
```

대형 프로그램의 개발


pgm.h
#includes ...
#defines ...
...
list of fct prototypes
main.c
#include "pgm.h"
...
fct.c
#include "pgm.h"
...
prn.c
#include "pgm.h"
...

대형 프로그램의 개발

```
/* pgm. h */

#include <stdio.h>
#include <stdlib.h>
#define   N   3

void    fctl(int k);
void    fct2(void);
void    wrt_info(char *);
```

대형 프로그램의 개발

```
/* main.c */
#include "pgm.h"
int main (void)
{
    char    ans;
    int     i, n = N;
    printf("%s" ,
        "This program does not do very much.\n"
        "Do you want more information?  ");
    scanf(" %c", &ans);
    if (ans == 'y' || ans == 'Y')
        wrt_info("pgm");
    for (i = 0; i < n; ++i)
        fct1(i);
    printf("Bye!\n");
    return 0;
}
```

대형 프로그램의 개발

```
/* fct.c */

#include "pgm.h"

void fct1(int n)
{
    int  i;
    printf("Hello from fct1()\n");
    for (i = 0; i < n; ++i)
        fct2();
}

void fct2(void)
{
    printf("Hello from fct2()\n");
}
```

대형 프로그램의 개발

```
/* wrt.c */
#include "pgm.h"

void wrt_info(char *pgm_name) {
    printf("Usage: %s\n\n", pgm_name);
    printf("%s\n",
    "This program illustrates how one can write a program\n"
    "in more than one file. In this example, we have a\n"
    "single .h file that gets included at the top of our\n"
    "three .c files. Thus, the .h file acts as the \"glue\"\n"
    "that binds the program together.\n"
    "\n"
    "Note that the functions fct1() and fct2() when called\n"
    "only say \"hello.\" When writing a serious program, the\n"
    "programmer sometimes does this in a first working\n"
    "version of the code.\n"); }
```

대형 프로그램의 개발

- 다음과 같이 컴파일할 수 있음

```
gcc -o pgm main.c fct.c prn.c
```

- 컴파일러는 세 개의 .c 파일을 컴파일 하고 세 개의 .o 파일로 구성된 하나의 실행 가능한 pgm파일을 만듦

단정

```
#include <assert.h>
#include <atdio.h>

int f(int a, int b);
int g(int c);

int main(void)
{
    int  a, b, c;
    ...
    scanf("%d%d", &a, &b);
    ...
    c = f(a, b);
    assert(c > 0);    /* an assertion */
    ...
}
```

단정

- assert()매크로에 인자로 전달된 관계식이 거짓 값을 갖는 경우 시스템은 메시지를 출력하고 프로그램을 중단시킴

(예)인자 a는 1이나 -1값을 가지고
b의 예상 구간을 [7, 11] 안에 있다고 가정할 경우

```
int f(int a, int b)
{
   ...
   assert(a == 1 || a == -1);
   assert(b >= 7 && b <= 11);
   ...
}
```

변수의 유효범위 규칙

- 유효범위 규칙: 변수나 상수가 유효하게 정의되는 범위
- 범위 효력이 정해지는 시점에 따라 정적 유효범위 규칙과 동적 유효범위 규칙으로 나뉜다.
  - → 동적 유효범위 : 실행시간에 결정 (실행 순서에 의해)
  - → 정적 유효범위 : 컴파일 시간에 결정 (프로그램의 구조에 의해)
- C에서는 정적 유효범위 규칙을 사용한다.

유효범위의 규칙

- 기본적인 유효범위 규칙은 변수가 선언된 블록 안에서만 그 변수를 이용할 수 있음
- 외부 블록의 변수는 내부에서 다시 정의하지 않는 한 여전히 유효함
- 만일 내부 블록에서 해당 식별자가 다시 정의되면, 외부 변수는 내부 블록으로부터 숨겨짐

5.10 Scope Rules

```
{
    int a = 2;              /* outer block a */
    printf("%d", a);        /* 2 is printed */
    {
        int a = 5;          /* inner block a */
        printf("%d", a);    /* 5 is printed */
    }                       /* back to the outer block */
    printf(%d", ++a);       /* 3 is printed */
}

{
    int a_outer = 2;
    printf("%d", a);
    {
        int a_inner = 5;
        printf("%d", a_inner);
    }
    printf(%d", ++a_outer);
}
```

5.10 Scope Rules

```
{
    int a = 1, b = 2, c = 3;
    printf("%3d%3d%3d", a, b, c);           /* 1 2 3 */
    {
        int b = 4;
        float c = 5.0;
        printf("%3d%3d%5.1f", a, b, c);     /* 1 4 5.0 */
        a = b;
        {
            int c;
            c = b;
            printf("%3d%3d%3d", a, b, c);   /* 4 4 4 */
        }
        printf("%3d%3d%5.1f", a, b, c);     /* 4 4 5.0 */
    }
    printf("%3d%3d%3d", a, b, c);           /* 4 2 3 */
}
```

기억영역 클래스 auto

- C언어에서 사용되는 변수는 두 가지 속성을 가진다. 하나는 형(type)이고 또 다른 하나는 기억영역 클래스(storage class)이다.
  - → auto extern register static
- 함수의 몸체 부분에서 선언된 변수는 디폴트로 auto(자동) 기억영역 클래스이다.
- 키워드 auto 를 사용하여 자동 기억영역 클래스임을 명시적으로 선언 할 수도 있다.
  - → auto int a, b, c;
  - → auto float f;
- 지역변수들은 블록을 빠져 나가게 되면 모두 메모리에서 제거되어 사용할 수 없게 된다. 그러므로 변수가 저장하고 있던 값들을 잃게 된다.

기억영역 클래스 extern

- 블록과 함수 간에 정보를 교환하기 위한 방법 중 외부 변수(extern)를 사용하는 방법이 있다.
- 변수를 함수 밖에서 선언하면, 프로그램이 종료할 때까지 메모리에 계속 남아 있다.
- 외부변수(external variable)들은 전역변수(global variables)로 취급되어 외부 변수 선언 뒤에 있는 블록이나 함수가 끝나더라도 사라지지 않고 변수로 존재하게 된다.

기억영역 클래스 extern

```
#include <stdio.h>
int     a = 1, b = 2, c = 3;        /* global variables */
int     f(void);                    /* function prototype */

int main(void)
{
    printf("%3d\n", f());           /* 12 is printed */
    printf("%3d%3d%3d\n", a, b, c); /* 4 2 3 is printed */
    return 0;
}

int f(void)
{
    int  b, c;                      /* b and c are local */
                                    /* global b, c are masked */
    a = b = c = 4;
    return (a + b + c);
}
```

기억영역 클래스 extern

```
/* file1.c */
#include <stdio.h>
int   a = 1, b = 2, c = 3;       /* external variables */
int   f(void);

int main(void)
{
    printf("%3d\n", f());
    printf("%3d%3d%3d\n", a, b, c);
    return 0;
}

/* file2.c */
int f(void)
{
    extern int a;                /* look for it elsewhere */
    int        b, c;
    a = b = c = 4;
    return (a + b + c);
}
```

기억영역 클래스 extern

- 위 예문의 두 파일은 분리되어 컴파일 될 수 있다.
- 두 번째 파일에서 extern 의 사용은 컴파일러에게 이 변수가 현재 파일 내의 어디서나 선언되어 있거나 또는 다른 파일 내에 선언되었음을 알려 준다.
- 외부변수는 프로그램 실행되는 동안 전체에 걸쳐 존재하기 때문에 함수들에 값을 전달하는 용도로 사용될 수 있다.
- 함수 안으로 정보를 전달하는 데는 두 가지 방법이 있다. 하나는 외부 변수를 사용하는 것이고, 다른 하나는 매개변수를 사용하는 방식이다.

기억영역 클래스 extern

- 함수 간에 정보를 전달하는 두 가지 방법
  - → 외부 변수
  - → 매개변수 메커니즘
- 외부 변수
  - → 사용이 용이
  - → 부작용이 발생할 가능성이 있음
- 매개변수 매커니즘
  - → 코드의 모듈성을 향상시킴
  - → 원하지 않는 부작용의 가능성을 줄일 수 있음
    - ▸ 전역 변수의 경우, 함수 내부에서 변경 가능

기억영역 클래스 register

- 기억영역 클래스 register 는 컴파일러에게 변수를 가능하다면 고속 메모리인 레지스터에 저장되도록 지시
- 한정된 자원으로 인해 할당하지 못하면, 이 기억영역 클래스는 디폴트로 자동 기억영역 클래스가 됨
- 이러한 변수는 사용되기 바로 전에 선언하는 것이 좋음

```
{
    register int i;
    for(i=0; i < LIMIT; ++i) {
        ...
    }
}       /* block exit will free the register */
```

기억영역 클래스 static

- 정적(static) 변수의 용도
  - 기본용도: 그 블록으로 다시 들어갈 때 지역 변수로 선언된 변수가 이전의 값을 유지하고 있게 하는 것
  - 외부 선언과 관련된 용도 (5.12 참조)

```
void f(void)
{
    static int   cnt = 0;
    ++cnt;
    if(cnt%2 == 0)
        ...   /* do something */
    else
        ...   /* do something different */
}
```

정적 외부 변수

- 정적 외부 구조물은 프로그램 모듈화에 있어서 매우 중요한 개념인 비공개를 제공함
- 비공개란 변수나 함수의 가시화 또는 유효범위의 제한을 의미함
- 정적 외부 변수의 유효범위는 자신이 선언되어 있는 원시 파일의 나머지 부분임
- 다른 파일에서 선언된 함수가 키워드 기억영역 클래스 extern을 사용하여 그 변수를 사용하고자 해도 사용할 수 없음

정적 외부 변수

```
void f(void)
{
   .....          /* v is not available here */
}

static int v;    /* static external variable */

void g(void)
{
   .....              /* v can be used here */
}
```

(예) 난수 발생기

```
#define     INITIAL_SEED        17
#define     MULTIPLIER          25173
#define     INCREMENT           13849
#define     MODULUS             65536
#define     FLOATING_MODULUS    65536.0

static unsigned seed = INITIAL_SEED; /* external, but */
                                     /* private to this file */
unsigned random(void)
{
    seed = (MULTIPLIER * seed + INCREMENT) % MODULUS;
    return seed;
}
double probability(void)
{
    seed = (MULTIPLIER * seed + INCREMENT) % MODULUS;
    return (seed / FLOATING_MODULUS);
}
```

(예) 난수 발생기

- randon()과 probability()는 호출될 때 마다 이전의 seed값을 이용해서 새로운 seed값을 생성
- seed는 정적 외부 변수이기 때문에 이 파일 안에서만 사용되고, 그 값은 함수 호출 사이에도 보존됨
- 이제 외부의 함수가 seed를 변경할 걱정없이 편리하게 seed를 활용할 수 있음

디폴트 초기화

- C언어에서 외부 변수와 정적 변수는 프로그래머가 초기화하지 않아도 시스템에 의해 0으로 초기화된다.
- 이와 같은 방식으로 초기화 되는 것에는 배열, 문자열, 포인터, 구조체, 공용체가 있다.
- 반면에, 자동 변수와 레지스터 변수는 일반적으로 시스템에 의해 초기화되지 않는다.
  - → 가비지 값을 가질 수도 있음

재귀호출

- 재귀호출(Recursive) : 함수가 자기 자신을 호출

(예)

```
#include <stdio.h>
int main(void)
{
   printf( " The universe is never ending! ");
   main();
   return 0;
}
```

재귀호출

- 단순한 재귀적 루틴은 일반적인 패턴을 따름
  - → 재귀의 일반적인 패턴에서는 기본적인 경우와 일반적인 재귀 경우를 처리하는 코드가 있음
  - → 보통 두 경우는 한 변수에 의해 결정됨
- 일반적인 재귀 함수의 제어 흐름
  1. 변수를 검사하여 기본적인 경우인지 일반적인 경우인지를 결정
  2. 기본적인 경우일 때에는 더 이상 재귀 호출을 하지 않고 필요한 값을 리턴
  3. 일반적인 경우일 때에는 그 변수의 값이 결국에 기본적인 경우의 값이 될 수 있게 하여 재귀 호출

(예) 1+2+..+n (양수의 합)

```
int sum(int n)
{
    if(n <= 1)
        return n;
    else
        return (n + sum(n - 1));
}


Function call   Value returned
sum(1)          1
sum(2)          2+sum(1) or 2+1
sum(3)          3+sum(2) or 3+2+1
sum(4)          4+sum(3) or 4+3+2+1
```

(예)

```
/* Write a line backwards. */
#include <stdio.h>
void wrt_it(void);
int main(void)
{
    printf("Input a line: ");
    wrt_it();
    printf("\n\n");
    return 0;
}
void wrt_it(void)
{
    int c;
    if ((c = getchar()) != '\n')
        wrt_it();   /* \n 전까지 계속 재귀호출 */
    putchar(c);     /* \n 들어오면 출력시작 */
}

Input a line : sing a song of sixpence¬
ecnepxis fo gnos a gnis
```

재귀호출의 효율성

- 많은 알고리즘은 재귀적 방식과 반복적 방식 둘 다로 표현할 수 있음
- 전형적으로, 재귀가 더 간결하고 같은 계산을 하는 데 더 적은 변수를 필요로 함
- 반면, 재귀는 각 호출을 위한 인자와 변수를 스택에 쌓아두어 관리하기 때문에 많은 시간과 공간을 요구함
- 즉, 재귀를 사용할 때에는 비효율성을 고려해야 함
- 그러나 일반적으로 재귀적 코드는 작성하기 쉽고, 이해하기 쉬우며, 유지보수하기가 쉬움

Homework

- Exercises #2, 3, 6, 11, 15, 22