5장 함수

함수

- 하향식 프로그래밍 기법
- 프로그램은 하나 이상의 함수로 구성됨
- 함수 정의
  - 함수가 수행할 일을 기술한 C 코드
- 함수 정의의 일반적인 형식

```
type function_name( parameter list )
{
    declarations
    statements
}
```

함수 헤더

- 헤더
  - 함수 정의에서 첫 번째 여는 중괄호의 앞 부분

  type function_name(parameter list)

함수 헤더

- *type function_name(parameter list)*
  - *type*
    - 함수가 리턴하는 값의 형
    - 컴파일러는 필요하다면, 함수의 리턴 값을 이 *type*으로 변환함
    - 이것이 void이면 리턴하는 값이 없다는 것을 나타냄
  - *parameter list*
    - 이 함수가 가지는 인자의 목록
    - 이 함수를 호출할 때에는 이 list에 맞게 호출해야 함
    - 이것이 void이면 인자를 갖지 않음을 나타냄

함수 몸체

- 몸체
  - 함수 정의에서 중괄호 사이에 있는 문장들
- 예제

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

함수 정의 및 호출

- 함수 정의

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

- 함수 호출

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

지역적/전역적 변수

- 지역 변수
  - 함수 몸체 안에서 선언된 변수
- 전역 변수
  - 함수 외부에서 선언된 변수

지역적/전역적 변수

```
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;
}
```

전통적인 C

- 전통적인 C에서는 매개변수 목록에 있는 변수들의 선언이 매개변수 목록과 첫번째 중괄호 사이에 옴

```
void f(a, b, c, x, y)
int      a, b, c;
double   x, y;
{
    .....
```

전통적인 C

- 매개변수가 없다면 빈 괄호만 써줌

```
void f1( )
{
        .....
```

return 문

- return 문을 만나면, 그 함수의 실행은 종료되고 제어는 호출한 환경으로 넘어감
- 만일 return 문이 수식을 포함하고 있으면, 그 수식의 값도 호출한 환경으로 같이 넘어감
- 또한 필요하다면, 그 수식의 값은 함수 정의에 명시된 함수의 형으로 변환됨

return 문

- 일반적인 형식

  return;

  return expression;

- 예

  return;

  return ++a;

  return (a * b);

return 문 예제

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

return 문 예제

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

함수 원형

- 새로운 함수 선언 구문
- 컴파일러에게 함수로 전달되는 인자의 수와 형 그리고 함수의 리턴 값의 형을 알려줌
- 일반적인 형식

  type function_name(parameter type list);

함수 원형

- *type function_name(parameter type list);*
  - *parameter type list* 에서 식별자의 사용은 옵션

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

  - 가변인자는 *parameter type list* 를 ...으로 표현

    ```
    printf(const char *, ...)
    ```

컴파일러 관점에서의 함수선언

- 컴파일러는 여러 방법으로 함수 선언을 인식함
  - 함수 호출
    - 만일 함수 선언이나 정의 전에 f(x)와 같은 함수 호출을 사용했다면 컴파일러는 디폴트로 다음과 같은 형태를 가정함

      ```
      int f();
      ```
  - 함수 정의
  - 함수 선언/함수 원형

함수 정의 순서

- main() 함수의 위치에 따라
  - main()을 다른 함수 보다 먼저 정의
    - 다른 함수를 사용할 수 있게 하기 위해 다른 모든 함수의 함수 원형을 main()앞에 선언해야 함
  - main()을 다른 함수 다음에 정의

함수 호출

- 프로그램은 하나의 main() 함수와 다른 함수들로 구성됨
- 함수 관점에서의 프로그램 수행
  - 프로그램은 항상 main() 함수부터 수행됨
  - 프로그램의 제어가 함수 이름을 만나면 그 함수가 호출됨
  - 함수가 호출되면 프로그램의 제어는 호출된 함수로 넘어가고 그 함수를 수행함
  - 호출된 함수가 실행을 완료하면 프로그램의 제어는 그 함수를 호출한 환경으로 다시 넘어가고, 제어를 다시 받은 함수는 자기 일을 계속 수행함

제어의 흐름

```
#include <stdio.h>
int test(int);
int main(void)
{
    int b, a;

    a = 5;
    b = test(a);
    printf("test is %d", b);
    return 0;
}
```

```
int test(int c)
{
    c = c + 10;
    return 1;
}

int printf( . . . )
{
    . . . . .
}
```

제어의 흐름

```c
#include <stdio.h>
int test(int);
int main(void)
{
    int b, a;

    a = 5;
    b = test(a);
    printf("test is %d", b);
    return 0;
}
```

```c
int test(int c)
{
    c = c + 10;
    return 1;
}

int printf( . . . )
{
    . . . . .
}
```

제어의 흐름

```
#include <stdio.h>
int test(int);
int main(void)
{
    int b, a;

    a = 5;
    b = test(a);
    printf("test is %d", b);
    return 0;
}
```

```
int test(int c)
{
    c = c + 10;
    return 1;
}

int printf( . . . )
{
    . . . . .
}
```

제어의 흐름

```c
#include <stdio.h>
int test(int);
int main(void)
{
    int b, a;

    a = 5;
    b = test(a);
    printf("test is %d", b);
    return 0;
}
```

```c
int test(int c)
{
    c = c + 10;
    return 1;
}

int printf( . . . )
{
    . . . . .
}
```

제어의 흐름

```
#include <stdio.h>
int test(int);
int main(void)
{
    int b, a;

    a = 5;
    b = test(a);
    printf("test is %d", b);
    return 0;
}
```

```
int test(int c)
{
    c = c + 10;
    return 1;
}

int printf( . . . )
{
    . . . . .
}
```

제어의 흐름

```c
#include <stdio.h>
int test(int);
int main(void)
{
    int b, a;

    a = 5;
    b = test(a);
    printf("test is %d", b);
    return 0;
}
```

```c
int test(int c)
{
    c = c + 10;
    return 1;
}

int printf( . . . )
{
    . . . . .
}
```

제어의 흐름

```
#include <stdio.h>
int test(int);
int main(void)
{
    int b, a;

    a = 5;
    b = test(a);
    printf("test is %d", b);
    return 0;
}
```

```
int test(int c)
{
    c = c + 10;
    return 1;
}

int printf( . . . )
{
    . . . . .
}
```

제어의 흐름

```
#include <stdio.h>
int test(int);
int main(void)
{
    int b, a;

    a = 5;
    b = test(a);
    printf("test is %d", b);
    return 0;
}
```

```
int test(int c)
{
    c = c + 10;
    return 1;
}

int printf( . . . )
{
    . . . . .
}
```

값에 의한 호출

- C에서는 값에 의한 호출로 함수를 호출함
  - 각 인자가 평가된 후, 그 값이 대응되는 형식 매개변수의 위치에서 지역적으로 사용됨
  - 변수가 함수로 전달되어도, 호출한 환경에 저장된 변수 값은 변경되지 않음

값에 의한 호출의 예

```
#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;
}
```

함수 호출의 의미

1. 인자 목록의 각 수식이 평가된다.
2. 필요하다면, 그 수식의 값이 형식 매개변수의 형으로 변환되고, 함수 몸체의 시작 부분에서 그 값이 대응되는 형식 매개변수에 할당된다.
3. 함수의 몸체가 실행된다.
4. return 문을 만나면, 제어는 호출한 환경으로 넘어간다.
5. return 문이 수식을 가지고 있다면, 필요한 경우 그 수식의 값이 함수의 형으로 변환된 다음 그 값도 호출한 환경으로 넘어간다.
6. return 문이 수식을 가지지 않는다면, 어떠한 유용한 값도 호출한 환경으로 리턴되지 않는다.
7. return 문이 없다면, 제어가 함수 몸체의 끝에 도달할 경우 호출한 환경으로 넘어간다. 이때 아무런 값도 리턴되지 않는다.
8. 모든 인자는 "값에 의한 호출"로 넘어간다.

대형 프로그램

- 큰 프로그램은 별도의 디렉토리에 .h 파일과 .c 파일로 작성됨
- .h 파일은 헤더 파일이라고 하며, 여기에는 프로그램 전체에서 필요한 프로그램 구성 원소인 `#define`, `#include`, 열거형의 틀, 구조체와 공용체의 틀, 다른 프로그래밍 구조물, 그리고 함수 원형들을 포함함
- .c 파일에는 함수들을 정의함
- 각 .c 파일의 제일 처음에 헤더파일을 include함

예제

- pgm.h 파일

```
#include <stdio.h>
#include <stdlib.h>
#define   N   3
void   fct1(int k);
void   fct2(void);
void   prn_info(char
*);
```

- main.c 파일

```
#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')
        prn_info("pgm");
    for (i = 0; i < n; ++i)
        fct1(i);
    printf("Bye!₩n");
    return 0;
}
```

예제

- fct.c 파일

```
#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");
}
```

- wrt.c 파일

```
#include "pgm.h"
void prn_info(char *pgm_name)
{
    printf("Usage:%s₩n₩n",
pgm_name);
    printf("%s₩n",
     "This program illustrates how
one can write a program₩n");
}
```

컴파일

- cc -o pgm main.c fct.c wrt.c

단정

- 단정은 프로그램을 정확히 작동하게 만들며, 프로그램의 작성 의도를 쉽게 이해할 수 있게 해줌
- 표준 헤더 파일 assert.h에 있는 assert() 매크로 사용
  - assert()에 인자로 전달된 수식이 거짓이면, 시스템은 메시지를 출력하고 프로그램을 중단시킴

단정

- 예제

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

유효범위 규칙

- 기본적인 유효범위 규칙
  - 식별자는 그 식별자가 선언된 블록 안에서만 이용이 가능하다

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

유효범위 규칙

- 외부 블록 이름은 내부 블록이 그것을 다시 정의하지 않는 한, 내부 블록에서도 유효함
- 만일 다시 정의된다면, 외부 블록 이름은 내부 블록으로부터 숨겨짐

유효 범위 규칙

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

병렬 블록과 중첩 블록

```
{
   int  a, b;
   .....
   {          /* inner block 1 */
      float   b;
      .....  /* int a is known, but not int b */
   }
   .....
   {          /* inner block 2 */
      float    a;
      .....  /* int b is known, but not int a */
             /* nothing in inner block 1 in known */
   }
   .....
}
```

디버깅을 위한 블록 사용

- 코드 부분에 임시로 블록을 삽입하면, 프로그램의 다른 부분에 영향을 주지 않는 지역 변수를 사용할 수 있음

```
{ /* debugging starts here */
  static int   cnt = 0;
  printf("*** debug : cnt = %d  v = %d\n",
  ++cnt, v);
}
```

기억영역 클래스

- C의 모든 변수와 함수는 두 가지 속성을 가짐
  - 형, 기억영역 클래스
- 4 가지 기억영역 클래스
  - 자동, 외부, 레지스터, 정적
- 대응되는 키워드
  - auto extern register static

기억영역 클래스 auto

- 함수의 몸체에서 선언된 변수는 디폴트로 자동
- 블록 안에서 선언된 변수는 묵시적으로 자동 기억영역 클래스임
- auto를 사용하여 기억영역 클래스를 명시할 수도 있지만, 보통은 사용하지 않음
- 블록을 들어갈 때, 자동 변수들을 위해 메모리가 할당되고,
- 블록을 빠져나갈 때, 자동 변수가 할당 받은 메모리는 회수됨

기억영역 클래스 extern

- 함수 밖에서 선언된 변수의 기억영역 클래스는 extern
- 이것은 프로그램이 종료될 때까지 메모리에 계속 남아 있게 됨
- 이러한 변수들을 외부 변수라고 함
- 블록들과 함수들 간에 정보를 전달하는 방법으로 사용
- 함수 밖에서 선언된 변수들은 키워드 extern을 사용하지 않아도 외부 기억영역 클래스를 가짐
- 외부 변수들은 자동이나 레지스터 기억영역 클래스를 가질 수 없음

기억영역 클래스 extern

```
#include <stdion.h>
int   a = 1, b = 2, c = 3;   /* global variables */
/* or extern int   a = 1, b = 2, c = 3;       */
int   f(void);                /* function prototype */
int main(void){
   printf("%3d\n", f());          /* 12 */
   printf("%3d%3d%3d\n", a, b, c);  /* 4  2  3 */
   return 0;
}
int f(void){
   int   b,   c;              /* b and c are local */
   a = b = c = 4;
   return (a + b + c);
}
```

기억영역 클래스 extern

- 키워드 extern은 컴파일러에게 "이 변수를 현재 파일이나 다른 파일에서 찾아라"라고 지시하는 것임

```
/* In file 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;
```

```
/* In file 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

- 함수 간에 정보를 전달하는 두 가지 방법
  - 외부 변수
  - 매개변수 메커니즘
- 외부 변수
  - 사용이 용이
  - 부작용이 발생할 가능성이 있음
- 매개변수 메커니즘
  - 코드의 모듈성을 향상시킴
  - 원하지 않는 부작용의 가능성을 줄일 수 있음

기억영역 클래스 register

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

기억영역 클래스 static

- 정적 선언은 두 가지 중요한 용도로 사용됨
  - 블록에서 선언된 변수가 프로그램이 끝날 때까지 그 값을 계속 유지하도록 함
  - 외부 선언과 관련된 용도 (정적 외부 변수 참조)

기억영역 클래스 static

- 예제

```
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

- 난수 발생기 예제

```
#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);
}
```

디폴트 초기화

- 외부 변수와 정적 변수는 프로그래머가 초기화하지 않아도 시스템에 의해 0으로 초기화됨
- 자동 변수와 레지스터 변수는 일반적으로 시스템에 의해 초기화되지 않음

재귀

- 어떤 함수가 직접이든 간접이든 자기 자신을 호출하는 것

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

재귀

- 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

재귀

- 예제 코드

```
int sum(int n){
    if (n <= 1)
        return n;                    /* 기본적인 경우 */
    else                              /* 일반적인 경우 */
        return (n + sum(n - 1));
}
```

  - 이 코드에서는 n을 사용하여 두 경우를 판단
  - n이 1보다 작거나 같으면 기본적인 경우임
    - n을 리턴
  - 아니면 일반적인 경우임
    - n에서 1을 빼어 재귀 호출 ==> n에서 1을 뺐기 때문에 언젠가 n은 1보다 작거나 같아 질 것임

재귀의 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