Functions

adopted from KNK C Programming : A Modern Approach

개요

- 함수는 하나의 이름으로 그룹된 연속된 여러 개의 문장들임.
- 각 함수는 마치 작은 프로그램처럼 동작함.
  - 각자가 독립적인 선언문과 문장들을 갖음

- 함수의 장점:
  - 하나의 프로그램은 이해와 수정이 편하도록 여러 조각으로 나뉠 수 있음
  - 한 번 이상 사용되는 코드를 반복적으로 다시 쓸 필요 없음
  - 다른 프로그램에서 활용된 함수를 또 다른 프로그램에 쉽게 이식할 수 있음

Defining and Calling Functions 정의와 호출

- 함수를 정식으로 정의하기 전에 함수를 정의하는 간단한 예를 살펴 봅시다

Program: 평균 계산하기, 예제 1

- average 라는 이름의 함수는 두 개의 double 형 값의 평균을 구함:

```
double average(double a, double b)
{
  return (a + b) / 2;
}
```

- 함수 이름 앞에 double 은 average 함수의 ***return type*** 을 정의함
- 식별자 a 와 b (함수의 파라미터***parameters***)는 average 이 호출되면 사용될 값을 나타냄

Program: Computing Averages

- 모든 함수는 실행 가능한 부분인 ***body바디*** 가 있고 괄호로 묶여 있음

- `average` 의 바디에는 하나의 `return` 문장이 있음.

- 이 문장을 실행하면 이 함수를 호출한 위치로 되돌아가며 `(a + b) / 2` 의 계산 결과과 리턴됨

Program: Computing Averages

- 함수 호출: 함수의 이름과 인자 ***arguments의 목록이*** 필요
  - `average(x, y)`가 실행되면 `average` 함수가 호출.

- 인자는 함수에 정보를 전달하는 역할을 함.
  - `average(x, y)` 호출하면 `x` 와 `y` 의 값이 함수 선언에 쓰인 파라미터 `a` 와 `b`에 복사됨
- 인자는 꼭 변수일 필요는 없음; 같은 형의 다른 수식도 가능함
  - `average(5.1, 8.9)` 이나 `average(x/2, y/3)` 등이 가능함

Program: Computing Averages

- `average` 의 호출이 있는 위치가 결과가 받아 활용할 위치

- `x` 와 `y`의 값을 받아 평균을 출력하는 문장의 예:

```
printf("Average: %g\n", average(x, y));
```

`average` 의 리턴 값은 저장되지 않고 출력된 후 버려짐

- 만약 결과를 활용할 예정이라면 변수에 할당할 수 있음:

```
avg = average(x, y);
```

Program: Computing Averages

- `average.c` 프로그램은 세 개의 값을 읽어서 `average` 함수를 사용하여 두 숫자씩 평균을 구함:

```
Enter three numbers: 3.5 9.6 10.2
Average of 3.5 and 9.6: 6.55
Average of 9.6 and 10.2: 9.9
Average of 3.5 and 10.2: 6.85
```

Program: Computing Averages 코드

average.c

```
/* Computes pairwise averages of three numbers */

#include <stdio.h>

double average(double a, double b)
{
  return (a + b) / 2;
}

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}
```

Program: 카운트 다운 출력하기, 예제 2

- 만약 함수에 리턴이 없다면, 그 사실을 알리기 위해서 리턴 타입을 `void`을 함수 이름 앞에 씀
  - `void` 는 값이 없음을 알리는 형이다.

```
void print_count(int n)
{
  printf("T minus %d and counting\n", n);
}
```

- `print_count` 호출은 독립적으로 실행되어야 함:

```
print_count(i);
```

- `countdown.c` 프로그램에서 `print_count` 를 루프 내에서 10 번 호출할 것임

countdown.c

```
/* Prints a countdown */

#include <stdio.h>

void print_count(int n)
{
  printf("T minus %d and counting\n", n);
}

int main(void)
{
  int i;

  for (i = 10; i > 0; --i)
    print_count(i);

  return 0;
}
```

Program: Printing a Pun (Revisited), 예제 3

- 함수에 파라미터가 없다면 함수 이름 뒤의 괄호 내에 `void` 을 쓰면 됨:

```
void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
}
```

- 인자 없이 함수를 호출하려면 함수이름과 괄호만 적으면 됨:

```
print_pun();
```

  괄호는 절대 있어야 함.

- `pun2.c` 프로그램은 `print_pun` 함수를 활용함.

pun2.c

```
/* Prints a bad pun */

#include <stdio.h>

void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
}

int main(void)
{
  print_pun();
  return 0;
}
```

함수의 정의

- 일반적인 형태의 ***function definition 함수 정의:***

return-type function-name (*parameters*)
{
    declarations
    statements
}

Function Definitions

- 함수의 리턴 타입은 함수가 종료되면서 되돌려줄 값의 형을 정의함
- 리턴 타입에 대한 규칙:
  - 함수로 배열 리턴하지 말자
  - 리턴 타입을 `void`으로 쓰면 이 함수는 어떤 값도 리턴하지 않음을 나타냄

- C89 표준에서 리턴 타입이 생략되면 함수의 리턴 타입이 `int`인 것으로 가정함
- C99 표준에서는 리터 타입을 생략할 수 없음.

Function Definitions

- 스타일에 따라 다르지만, 어떤 사람들은 리턴 타입을 함수 이름 위에 쓰기도 함:

```
double
average(double a, double b)
{
  return (a + b) / 2;
}
```

- 리턴 타입이 매우 긴 경우 분리해서 쓰는 것도 현명한 방법임
  - 예: `unsigned long int`.

Function Definitions

- 함수 이름 뒤에 파라미터(매개 변수)의 목록이 따름.
- 각 파라미터마다 형이 명시되어야 하며 파라미터들은 쉼표로 분리됨
- 파라미터가 없는 함수에는 `void` 라고 괄호 내에 적어야 함

Function Definitions

- 함수 바디에는 선언문과 문장들을 씀

- 앞에서 정의했던 `average` 함수를 다시 작성한 예:

```
double average(double a, double b)
{
  double sum;          /* declaration */

  sum = a + b;         /* statement */
  return sum / 2;      /* statement */
}
```

Function Definitions

- 한 함수 내에서 사용되는 변수들은 다른 함수에 의해 변경 및 참조 될 수 없음
- C89표준에서는 변수 선언이 다른 어떤 문장보다 우서해야 함
- C99표준에서는 선언된 변수가 사용하는 문장보다 앞에 있기만 하면 변수 선언과 문장이 혼재 될 수 있음

Function Definitions

- 함수의 리턴 값이 `void` ("`void` 함수"라함) 인 경우 바디가 없을 수 있음:

```
void print_pun(void)
{
}
```

- 프로그램을 작성하는 과정에서 임시적으로 바디를 작성하지 않은 경우라고 보면 됨

Function Calls 함수 호출

- 함수 호출을 하려면 함수 이름을 쓰고 괄호 내에 인자들의 목록을 써야 함:

```
average(x, y)
print_count(i)
print_pun()
```

- 괄호가 없다면 호출 되지 않음:

```
print_pun;    /*** WRONG ***/
```

이런 문장을 쓸 수는 있지만, 아무런 효과가 없음.

Function Calls

- `void` 함수는 언제나 세미콜론을 사용하여 하나의 문장으로 만들어야 함:

```
print_count(i);
print_pun();
```

- 비`void` 함수는 결과가 있기 때문에, 결과를 저장할 변수가 있어야 함. 비교 출력 등의 작업에 활용할 수 있음:

```
avg = average(x, y);
if (average(x, y) > 0)
  printf("Average is positive\n");
printf("The average is %g\n", average(x, y));
```

Function Calls

- 비`void` 함수의 리턴 값은 필요에 따라 언제든지 버려질 수 있음:

```
average(x, y);  /* discards return value */
```

표현식으로 함수가 활용된 예로서, 계산은 했지만, 그 결과는 버려졌음

Function Calls

- `average` 의 리턴 값을 무시하는 것은 이상해 보일 수 있지만, 다른 함수들에서는 그런 활용이 이해가 되기도 함
- `printf` 의 리턴 값은 출력된 문자의 수를 나타냄.
- 다음과 같은 호출 문의 결과로 `num_chars` 는 9를 갖음:

```
num_chars = printf("Hi, Mom!\n");
```

- 일반적으로 `printf`의 리턴 값을 버림:

```
printf("Hi, Mom!\n");
  /* discards return value */
```

Function Calls

- 명시적으로 어떤 함수의 리턴 값을 무시하겠다는 표현으로 함수 호출문 앞에 `(void)` 을 쓰기도 함I:

```
(void) printf("Hi, Mom!\n");
```

- `(void)` 을 쓰면 코드를 읽는 사람이 리턴 값이 있지만 버리기로 했다는 것을 명확히 알 수 있음

Program: 소수인지 판별 프로그램

- `prime.c` 는 소수를 판별하는 프로그램:

```
Enter a number: 34
Not prime
```

- 이 프로그램은 `is_prime` 이름의 함수를 사용함. 결과로 소수이면 `true`, 아니면 `false`를 리턴함.

- `is_prime` 는 입력받은 파라미터 `n` 을 2 부터 `n`의 제곱근까지의 수로 나눔; 한 번이라도 나머지가 0이면 그 수는 소수가 아님

prime.c

```
/* Tests whether a number is prime */

#include <stdbool.h>   /* C99 only */
#include <stdio.h>

bool is_prime(int n)
{
  int divisor;

  if (n <= 1)
    return false;
  for (divisor = 2; divisor * divisor <= n; divisor++)
    if (n % divisor == 0)
      return false;
  return true;
}
```

```
int main(void)
{
  int n;

  printf("Enter a number: ");
  scanf("%d", &n);
  if (is_prime(n))
    printf("Prime\n");
  else
    printf("Not prime\n");
  return 0;
}
```

Function Declarations 함수 선언

- 함수의 정의는 함수의 호출 전에 있을 필요는 없음
  - 선언과 정의는 서로 다름!!

- `average.c` 프로그램의 순서를 바꾸어 `average` *정의를* `main` 함수 정의 뒤에 작성해보자

Function Declarations

```
#include <stdio.h>

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}

double average(double a, double b)
{
  return (a + b) / 2;
}
```

Function Declarations

- 컴파일러가 `main` 에서 처음으로 `average` 호출을 만났다고 하면 해당 정보가 없음.

- 에러 메시지를 출력하기 전, 컴파일러는 호출문의 문장을 보고 `average` 함수가 `int` 결과를 리턴한다는 것을 알아냄.

- 이 때, 컴파일러가 함수를 ***implicit declaration*** 암묵적으로 선언했다고 함.

Function Declarations

- 앞의 경우 컴파일러는 `average` 가 몇 개의 인자를 받아야 하는지 알 수 없으며 어떤 형을 가져야 하는지도 알 수 없음

- 기본 형을 가정하고, 그 결정이 맞기를 기대함

- 실제 `average` 의 정의를 프로그램에서 만나면 기대한 `int` 형이 아니라 `double`인 것을 알게 되기 때문에 오류 메시지를 출력하게 됨

Function Declarations

- 이와 같은 정의 전에 호출하는 문제를 해결하려면 호출문 보다 정의가 먼저오도록 함수들을 배치해야 함

- 하지만 항상 그런 배치를 하는 쉽지 않음

- 설령 그렇게 배치를 했다손치더라도 함수들의 순서가 읽기 어려운 순서로 되어 있을 수 있음

Function Declarations

- 이를 해결하기 위한 방법을 함수의 선언만 먼저 하도록 지원함

- ***function declaration 함수 선언***을 통해 컴파일러에게 나중에 실제 정의될 함수가 어떤 형태를 갖는지 미리 알려줌

- 함수 선언의 일반적인 형태:

  return-type function-name (*parameters*) ;

- 주의: 함수 선언과 정의는 같은 꼴이어야 함

Function Declarations

- //average.c 프로그램을 수정하여 average 선언문을 추가한 예

```
#include <stdio.h>

double average(double a, double b);    /* DECLARATION */

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}

double average(double a, double b)     /* DEFINITION */
{
  return (a + b) / 2;
}
```

Function Declarations

- 함수 선언은 ***function prototypes 함수 프로토타입***이라고 함.

- 함수의 프로토타입은 함수의 매개변수의 이름을 꼭 명시하지 않아도 됨. 대신 어떤 형의 변수를 쓸지는 꼭 명시해야 함:

```
double average(double, double);
```

- 파라미터 이름을 항상 쓰는 것이 좋음

Function Declarations

- C99표준에서는 함수의 선언 또는 정의가 그 함수의 호출 전에 꼭 나오는 것을 강제함

- 미리 선언 또는 정의되지 않은 함수를 호출하면 오류가 남

Arguments 인자

- c에서는 인자들은 ***passed by value 값으로 전달됨:*** 함수가 호출되면 각 인자가 계산되고 그 결과를 파라미터/매개변수에 할당함

- 매개변수가 인자의 값의 복사본을 갖기 때문에 매개변수를 변경하더라도 인자의 값은 변경되지 않음

Arguments

- 인자가 값으로 전달된다는 것에 장점과 단점이 있음
- 인자의 값을 변경하지 않으면서도 매개변수를 활용할 수 있기 때문에, 매개변수를 함수 내에서 변수로 쓸 수 있음.
  - 함수 내에 필요한 변수의 수를 줄이는 효과가 있음

Arguments

- 다음 함수는 x 를 n승하는 함수임:

```
int power(int x, int n)
{
  int i, result = 1;

  for (i = 1; i <= n; i++)
    result = result * x;

  return result;
}
```

Arguments

- `n`은 실제 지수의 *복사본*이기 때문에 `i`를 쓰지 않고 `n`을 변경해도 됨

```
int power(int x, int n)
{
  int result = 1;

  while (n-- > 0)
    result = result * x;

  return result;
}
```

Arguments

- c가 값으로 인자를 전달하기 때문에 어떤 종류의 함수들은 작성하기가 어려움
- 예를 들어 어떤 함수가 전달 받은 `double` 값을 실수부와 소수부로 구분해야 한다고 해보자

- 함수는 두 개의 수를 리턴할 수 없으므로 몇 개의 인자를 더 전달하여 그 변수를 변경하도록 해보자:

```
void decompose(double x, long int_part,
               double frac_part)
{
  int_part = (long) x;
  frac_part = x - int_part;
}
```

Arguments

- 함수 호출 문장:

```
decompose(3.14159, i, d);
```

- 안타깝게도, `i` 와 `d` 는 함수내에서 일어나는 `int_part` 와 `frac_part` 할당문에 의해 영향을 받지 않음
- 교재 11장에 `decompose` 가 제대로 동작하려면 어떻게 해야 하는지 설명됨.

Argument Conversions 인자 변환

- c 의 함수 호출은 매개 변수의 형과 인자의 형이 똑같지 않은 것을 허용함

- 단, 함수 호출 전에 컴파일러가 함수의 프로토타입이나 정의가 있는지에 따라 가능 여부가 결정됨

Argument Conversions

- ***호출 전에 컴파일러가 프로토타입을 만난 경우***

- 각 인자의 값은 암묵적으로 변환이 됨.
  - 할당할 때 값이 변환되는 것과 같음

- 예: 함수의 인자로 `double`을 기대하는 데 `int` 형 인자를 만나면, 인자의 형이 `double`로 바뀜.

Argument Conversions

- ***호출 전에 컴파일러가 프로토타입을 만나지 않은 경우***

- 컴파일러는 인자의 형을 기본 형으로 변환함*:*
  - `float` 형은 `double`로 변환
  - `char` 과 `short` 형 인자는 `int`으로 변환 (C99표준)

Argument Conversions

- 하지만, 이런 기본 변환 규칙을 따르는 것은 위험함.
- 예:

```
#include <stdio.h>

int main(void)
{
  double x = 3.0;
  printf("Square: %d\n", square(x));

  return 0;
}

int square(int n)
{
  return n * n;
}
```

- `square`가 호출되었을 때는 컴파일러가 함수의 매개변수의 형이 `int`인지 알지 못함

Argument Conversions

- 기본 변환 규칙에 따라 `x`를 변환하지만 효과가 없음.
- 인자의 형이 `int` 이기를 함수는 기대하지만 기본 변환 규칙에 따라 `double` 값을 갖고 있기 때문에 `square` 호출은 정의되지 않음
- 이런 경우에는 `square`의 인자를 올바른 형으로 변환하는 캐스팅을 하면 됨:

```
printf("Square: %d\n", square((int) x));
```

- 더 좋은 해법은 `square` 을 호출하기 전에 프로토타입을 먼저 쓰면 됨.

- C99 표준: `square` 을 선언이나 정의 전에 호출하면 오류가 남

Array Arguments 배열을 인자로 쓰기

- 함수의 매개변수가 일차원 배열인 경우, 배열의 길이를 명시하지 않아도 됨:

```
int f(int a[])  /* no length specified */
{
  …
}
```

- C는 전달받은 배열의 실제 길이를 판단할 방법을 제공하지 않음
  - 개발자가 길이를 또 다른 인자로 전달해야 함

Array Arguments

- 예:

```
int sum_array(int a[], int n)
{
  int i, sum = 0;

  for (i = 0; i < n; i++)
    sum += a[i];

  return sum;
}
```

- `sum_array`는 배열 `a`의 길이를 알아야 하기 때문에 길이를 두 번째 인자로 전달함

Array Arguments

- `sum_array`의 프로토타입은 다음과 같음:

```
int sum_array(int a[], int n);
```

- 프로토타입에서 함수가 쓸 매개변수의 이름은 역시 생략 가능함

```
int sum_array(int [], int);
```

Array Arguments

- `sum_array`가 호출되면 첫 인자는 배열의 이름이고, 두 번째 인자는 길이가 됨:

```
#define LEN 100

int main(void)
{
  int b[LEN], total;
  …
  total = sum_array(b, LEN);
  …
}
```

- 주의: 배열에 인덱스 표시를 위한 괄호는 생략함:

```
total = sum_array(b[], LEN);   /*** WRONG ***/
```

Array Arguments

- 함수는 배열의 길이를 파악할 방법이 전혀 없음
- 이 사실을 응용해 배열의 실제 길이보다 작게 배열을 길이를 나타내는 수를 인자로 쓸 수 있음

- 예를 들어 길이 100인 배열 `b`에 값을 50개만 저장했다고 해보자
- 다음과 같이 50개의 값만 처리하도록 쓸 수 있음

```
total = sum_array(b, 50);
```

Array Arguments

- 단, 배열의 실제 길이보다 더 큰 수를 전달하면 안됨:

```
total = sum_array(b, 150);    /*** WRONG ***/
```

`sum_array`는 실제 배열 길이보다 더 많은 수를 읽으려고 시도할 것이고 이는 정의되지 않은 동작을 만들어 냄

Array Arguments

- 함수가 매개변수로 배열을 받은 경우, 그 배열의 요소를 변경할 수 있음.
  - 변수와는 다르게 이 때는 인자의 요소도 변경됨.

- 요소의 값을 0으로 채우는 함수의 예:

```
void store_zeros(int a[], int n)
{
  int i;

  for (i = 0; i < n; i++)
    a[i] = 0;
}
```

Array Arguments

- `store_zeros` 호출의 예

```
store_zeros(b, 100);
```

- C는 인자를 값으로 전달한다고 했는데, 배열에 대한 설명과 모순됨
  - 교재 12장에 이런 모순에 대한 설명을 함

Array Arguments

- 매개변수가 다차원 배열인 경우 1차원의 길이만 생략 가능함.

- `sum_array`을 배열 `a`가 이차원인 경우로 수정해보자.
- 이 때 배열의 칸수를 지정해야 함:

```
#define LEN 10

int sum_two_dimensional_array(int a[][LEN], int n)
{
  int i, j, sum = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < LEN; j++)
      sum += a[i][j];

  return sum;
}
```

Array Arguments

- 임의의 칸수로 다차원 배열을 인자로 쓸 수 없다는 것은 불편할 수 있음
- 이런 어려움을 포인터의 배열이라는 개념으로 해결할 수 있음.

- C99표준의 가변 길이 배열을 매개변수로 쓰는 것이 더 좋음

가변 길이 배열 매개변수 (C99)

- C99 는 가변 길이 배열을 매개변수로 쓸 수 있도록 함
- `sum_array` 함수를 보자:

```
int sum_array(int a[], int n)
{
  …
}
```

  현재는 `n` 과 배열 `a`의 길이와는 관계가 없어 보임
- 함수 바디에서 `n`을 `a`의 길이로 다루겠지만 배열의 실제 길이는 `n`보다 크거나 작을 수 있음

Variable-Length Array Parameters (C99)

- 가변 길이 배열 매개변수를 쓰는 경우 `a`의 길이가 `n`이라고 명시할 수 있음:

```
int sum_array(int n, int a[n])
{
  …
}
```

- 첫 매개변수`n`은 두 번째 매개변수`a`의 길이를 명시함.

- 매개변수의 작성 순서가 바뀐 것에 유의해야 함
  - 가변 길이 배열 매개변수를 쓸 경우 순서가 중요함

Variable-Length Array Parameters (C99)

- 새로운 버전의 `sum_array`의 프로토타입을 작성하는 방법이 여럿 있음
- 한 가지 방법은 함수 정의 똑같이 쓰는 것:

```
int sum_array(int n, int a[n]);  /* Version 1 */
```

- 또 다른 방법은 배열의 길이를 `*`별표로 나타내는 것:

```
int sum_array(int n, int a[*]);  /* Version 2a */
```

Variable-Length Array Parameters (C99)

- `*`별표로 나타내는 이유는 매개변수의 이름이 함수 선언에서 선택적이기 때문.
- 만약 첫 매개변수가 생략되면, 두 번째 매개변수의 배열의 길이가 `n`인 것을 명시할 수 없지만 `*` 표를 써서 앞에 쓰인 매개변수가 배열의 길이인 것을 알릴 수 있음:

```
int sum_array(int, int [*]);     /* Version 2b */
```

Variable-Length Array Parameters (C99)

- 배열 매개변수에 빈 괄호를 쓰듯이 다음과 같이 빈 괄호도 허용됨:

```
int sum_array(int n, int a[]);   /* Version 3a */
int sum_array(int, int []);      /* Version 3b */
```

- 하지만, `n` 과 `a`의 관계가 표현되지 않기 때문에 괄호 안을 비워두는 것은 좋지 않음.

Variable-Length Array Parameters (C99)

- 일반적으로 가변 길이 배열 매개변수는 어떤 표현식도 가능

- 두 배열 `a` 와 `b`를 연접하여 세 번째 변수`c`에 저장하는 함수 예:

```
int concatenate(int m, int n, int a[m], int b[n],
                int c[m+n])
{
  …
}
```

- 배열 `c` 의 길이는 다른 두 개의 매개변수의 합으로 표현되고 있음.
  - 일반적으로는 함수 밖의 다른 변수이거나 또 다른 함수일 수도 있음

Variable-Length Array Parameters (C99)

- 일차원 가변 길이 배열 매개변수는 제한된 용법이 있음
- 함수 선언이나 정의에 이것이 사용되면 배열 인자의 길이를 한정 짓는 역할을 함
- 하지만, 다른 어떤 오류 검사를 하지 않음
  - 길이가 너무 길거나 짧은 오류가 있을 수 있음

Variable-Length Array Parameters (C99)

- 가변 길이 배열 매개변수는 다차원 배열에서 가장 유용함

- 가변 길이 배열 매개변수를 쓰면 아래의 함수에서 `sum_two_dimensional_array` 함수를 컬럼의 크기에 상관없도록 일반화 시킬 수 있음:

```
int sum_two_dimensional_array(int n, int m, int a[n][m])
{
  int i, j, sum = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
      sum += a[i][j];

  return sum;
}
```

Variable-Length Array Parameters (C99)

- 이 함수의 프로토타입은 다음과 같이 작성 가능함:

```
int sum_two_dimensional_array(int n, int m, int a[n][m]);
int sum_two_dimensional_array(int n, int m, int a[*][*]);
int sum_two_dimensional_array(int n, int m, int a[][m]);
int sum_two_dimensional_array(int n, int m, int a[][*]);
```

배열 매개변수 선언에 `static` 사용 (C99)

- C99표준은 `static` 키워드를 사용하여 배열 매개변수를 선언 가능함
- 다음의 예를 참고
  - `static` 을 사용하여 `a` 의 길이가 최소한 3임을 보장함:

```
int sum_array(int a[static 3], int n)
{
  …
}
```

Using **`static`** in Array Parameter Declarations (C99)

- `static` 의 사용 여부는 프로그램 동작에 영향 없음
- `static` 는 배열을 사용하는데 있어 좀 더 빠른 명령을 쓰도록 하는“힌트”로만 쓰일 뿐
- 다차원의 배열을 매개변수로 쓸 경우, `static` 은 일차원 첨자에만 쓸 수 있음

Compound Literals 복합 리터럴 (C99)

- `sum_array` 함수를 다시 살표 보자
- `sum_array` 가 호출되면 첫 번째 인자는 배열의 이름을 알림
- 예:

```
int b[] = {3, 0, 3, 4, 1};
total = sum_array(b, 5);
```

- `b` 는 변수로 호출되기 전에 선언되어야 함
- `b` 가 함수 호출이외에는 전혀 쓸모가 없다고 하면, 미리 선언하는 것 자체가 낭비 일 수 있음

Compound Literals (C99)

- C99표준에서는 ***compound literal*** 복합 리터럴(상수)을 사용하여 회피할 수 있음: 이름 없는 배열을 즉석에서 만들고 그 안에 포함될 요소들만 지정할 수 있음.
- `sum_array` 복합 리터럴을 사용하는 예 (볼드 처리됨) :

```
total = sum_array((int []){3, 0, 3, 4, 1}, 5);
```

- 배열의 길이가 선언될 때 쓰이지 않았기 때문에, 요소의 수를 세서 결정함
- 길이를 명시적으로 지정하는 방법:

```
(int [4]){1, 9, 2, 1}
```

아래와 동치

```
(int []){1, 9, 2, 1}
```

Compound Literals (C99)

- 복합 리터럴은 캐스트와 초기화가 복합적으로 쓰인 것과 같음
- 복합 리터럴과 초기화는 같은 규칙을 따름

- 복합 리터럴에서도 위치 지정 초기화를 할 수 있음

- 예를 들어, `(int [10]){8, 6}` 은 10개의 요소를 생성하고, 첫 두 개 요소의 값은 8과 6이며 나머지는 0으로 채움

Compound Literals (C99)

- 함수 호출문 내에서 생성된 복합 리터럴은 상수 외에도 다양한 수식을 받음:

```
total = sum_array((int []){2 * i, i + j, j * k}, 3);
```

- 복합 리터럴은 lvalue이기 때문에 요소들이 변경 가능함
- "읽기 전용"으로 만들려면 `const` 를 형 앞에 명시하면 됨:

```
(const int []){5, 4}
```

return 문

- void 형이 아닌 함수는 return 문을 써서 리턴할 값을 명시해야 함
- return 문의 용법

  return *expression* ;

- 표현식은 주로 상수 또는 변수임:

```
return 0;
return status;
```

- 좀 더 복잡한 표현식도 사용 가능함:

```
return n >= 0 ? n : 0;
```

The **`return`** Statement

- 만약 `return` 문과 리턴 형이 서로 다르면 암묵적으로 리턴 형으로 변환됨

- 함수의 리턴 형이 `int`이고 함수의 `return` 문이 `double` 표현식이라면, 표현식은 `int` 형으로 변함

The **return** Statement

- 리턴 형이 `void`인 함수에 `return` 문을 쓰려면 빈 표현식을 다음처럼 써야 함:

```
return;  /* return in a void function */
```

- 예:

```
void print_int(int i)
{
  if (i < 0)
    return;
  printf("%d", i);
}
```

The **return** Statement

- return 문이 void 함수 끝에 나올 수 있음:

```
void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
  return;   /* OK, but not needed */
}
```

  사실, return 문은 불필요함

- 비void 함수가 값을 전달하려고 시도하였는데 return 문의 실행을 실패하는 경우, 프로그램의 동작은 정의되어 있지 않음

프로그램 종료

- 일반적으로 `main` 함수의 리턴 형은 `int`:

```
int main(void)
{
  …
}
```

- 예전의 C 프로그램은 `main`의 리턴 형은 삭제 가능 함. 이 때 기본적으로 `int`형이 부여됨

```
main()
{
  …
}
```

Program Termination

- C99에서 함수의 리턴 형을 삭제 가능하긴 하지만, 삭제하지 않은 것을 권고함

- `main`의 매개변수 리스트에 매개변수가 없다면 `void`를 쓰지 않아도 괜찮음. 단, 쓰는 것을 권함

Program Termination

- `main`이 리턴하는 값은 프로그램 종료 시 상태 코드임
- 프로그램이 정상 종료한다면 `main`은 0을 리턴 함
- 비 이상적 종료하는 경우 `main` 함수는 0이외의 값을 리턴하도록 함.
- C 프로그램이 종료할 때 리턴문을 통해 상태 값을 리턴하는 습관을 들이는 것이 좋음

exit 함수

- 프로그램을 종료하는 방법 중 하나는 main 에 return 문을 쓰면 됨
- 또 다른 방법은 exit 함수를 호출하는 것(<stdlib.h>에 포함되어 있음)

- exit 에 전달하는 인자는 main의 리턴 값과 같은 역할을 함
- 정상 종료를 알리려면 0을 씀:

```
exit(0);    /* normal termination */
```

The **exit** Function

- 0 이라고 쓰면 직관적이지 않기 때문에 EXIT_SUCCESS 을 쓰기도 함(효과는 같음):

```
exit(EXIT_SUCCESS);
```

- EXIT_FAILURE은 비정상 종료를 뜻함:

```
exit(EXIT_FAILURE);
```

- EXIT_SUCCESS 와 EXIT_FAILURE 는 매크로로 <stdlib.h>에 정의됨
- EXIT_SUCCESS 와 EXIT_FAILURE 의 값은 C 컴파일러의 구현에 따라 차이가 있지만 일반적인 값은 0 과 1임

The **`exit`** Function

- 다음과 같이 쓴 문장은

  `return` *expression*`;`

  다음과 같이 쓴 것과 같음

  `exit(`*expression*`);`

- `return` 과 `exit` 차이는 `exit`는 어떤 함수 안에서 호출하든 프로그램이 종료된다는 것
- `return` 문은 `main` 함수 안에서만 프로그램을 종료하게 함.

Recursion 재귀

- 어떤 함수가 자기 자신을 부르면 ***recursive 재귀적***이라고 함.

- 다음 함수는 $n!$ 을 계산하기 위해 $n! = n \times (n-1)!$식을 이용해 재귀적으로 해결함

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

Recursion

- 재귀문이 동작하는 방식을 한 번 따라 가봅시다

```
i = fact(3);
```

fact(3) 3 은 1보다 작거나 같지 않기 때문에 다음을 호출함

fact(2), 2는 1보다 작거나 같지 않기 때문에 다음을 호출함

fact(1), 1이 기 때문에 1을 리턴함. 그리고

fact(2) 는 2 × 1 = 2 을 리턴하고. 그리고

fact(3) 은 3 × 2 = 6을 리턴함

Recursion

- 다음의 재귀 함수는 x^n을 하기 위해 다음 함수를 활용함

  $x^n = x \times x^{n-1}$.

```
int power(int x, int n)
{
  if (n == 0)
    return 1;
  else
    return x * power(x, n - 1);
}
```

Recursion

- power 함수를 계산하기 위해 조건부 표현식을 return 문에 쓸 수 있음:

```
int power(int x, int n)
{
  return n == 0 ? 1 : x * power(x, n - 1);
}
```

- Fact와 power 는 “종료 조건”을 주의하여 작성해야 함
- 모든 재귀함수는 무한 재귀문이 되지 않도록 종료조건이 꼭 필요 함

Option

재귀함수를 이용한 퀵정렬

재귀문의 예: 퀵정렬 알고리즘

- 재귀함수는 자기 자신을 호출해야 하는 복잡한 알고리즘에 유용하게 사용됨

- 재귀함수는 주로 ***divide-and-conquer 분할정복*** 이라는 알고리즘 기법에 주로 사용됨. 큰 문제의 크기를 작게 해서 같은 알고리즘으로 해결함

The Quicksort Algorithm

- 분할 정복 기법의 대표적인 예는 ***Quicksort*** 퀵소트 알고리즘이 있음
- 배열의 요소들이 1 부터 n까지 정렬되어 있다고 가정함

퀵정렬 알고리즘

1. *e*를 ("파티션 요소"라고 함) 선택하고 e를 중심으로 재정렬하는데 왼쪽은 e 보다 적거나 같은 요소들 1, …, $i-1$ 을 배치하고 오른쪽으로 e보다 크거가 같은 요소들 $i+1$, …, n 배치함.
2. 왼쪽 1, …, $i-1$ 을 퀵정렬로 재귀적으로 정렬함.
3. 오른쪽 $i+1$, …, n 을 퀵정렬로 재귀적으로 정렬함.

The Quicksort Algorithm

- 퀵정렬 알고리즘의 스텝 1 이 핵심

- 배열을 파티션하는 여러 방법이 있음
  - 그 중 효율적이진 않지만 구현이 쉬운 기법을 소개함

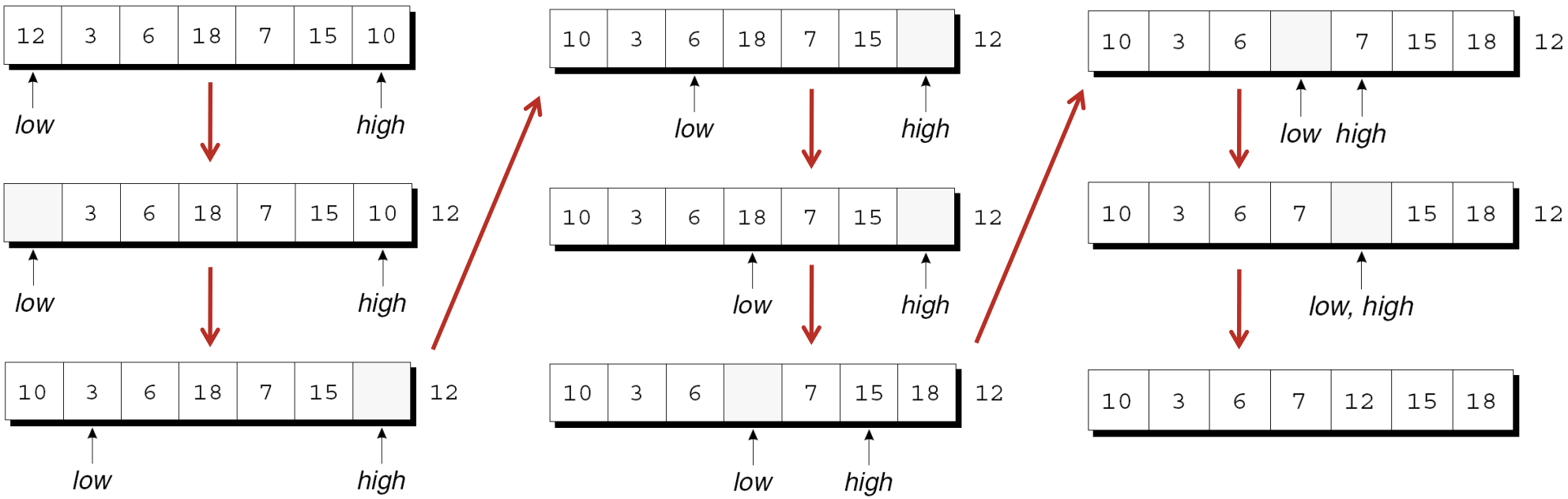
- 이 알고리즘은 *low* 와 *high* 라는 변수를 써서 배열에서의 위치를 파악함

The Quicksort Algorithm

- 최초에 *low* 는 첫 요소를 *high* 는 마지막 요소를 가리킴.
- 첫 요소(파티션 용 요소)를 임시 공간에 저장하여 배열에 구멍을 만듬
- 다음으로 *high* 가 가리키는 위치를 오른쪽에서 왼쪽으로 이동하면서 해당 위치의 값을 파티션 용 요소보다 작은 값을 만날 때까지 비교함
- 작은 값을 만나면 그 값을 비어 있는 첫 요소 위치로 이동함 (이 때 이동하면서 *high* 가 구멍이 됨)
- 이제 , *low* 를 왼쪽에서 오른쪽으로 이동하면서 파티션 용 요소보다 더 큰 값을 찾음. 찾게 되면 *high*가 가리키는 빈 구멍으로 옮김
- 이 과정은 *low* 와 *high* 가 가리키는 빈 구멍이 같을 때까지 반복됨
- 끝으로 그 구멍에 파티션 용 요소를 채움

The Quicksort Algorithm

- 배열의 파티션하기 예:



The Quicksort Algorithm

- 마지막 그림에서 파티션 용 요소 기준 왼쪽의 모든 요소들은 12보다 작거나 같고 오른쪽은 12보다 크거나 같음

- 배열이 파티션되어 있기 때문에, 재귀적으로 퀵정렬을 두 개의 배열을 인자로 하여 다시 호출함
  - 첫 번째 퀵정렬 함수를 호출할 때 배열의 요소들은 (10, 3, 6, and 7)
  - 두 번째 퀵정렬 함수를 호출 할 때 배열의 요소들은 (15 and 18).

Program: Quicksort

- 재귀함수 `quicksort` 를 만들어서 정수 형 배열을 퀵정렬로 정렬해보자
- 이 프로그램은 10개의 수를 읽어 들여 배열에 저장하고 `quicksort` 를 써서 정렬한 후 그 값들은 출력 함:

```
Enter 10 numbers to be sorted: 9 16 47 82 4 66 12 3 25 51
In sorted order: 3 4 9 12 16 25 47 51 66 82
```

- 배열을 파티션하는 함수는 `split`.

qsort.c

```
/* Sorts an array of integers using Quicksort algorithm */

#include <stdio.h>

#define N 10

void quicksort(int a[], int low, int high);
int split(int a[], int low, int high);

int main(void)
{
  int a[N], i;

  printf("Enter %d numbers to be sorted: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &a[i]);
  quicksort(a, 0, N - 1);

  printf("In sorted order: ");
  for (i = 0; i < N; i++)
    printf("%d ", a[i]);
  printf("\n");

  return 0;
}
```

```
void quicksort(int a[], int low, int high)
{
  int middle;

  if (low >= high) return;
  middle = split(a, low, high);
  quicksort(a, low, middle - 1);
  quicksort(a, middle + 1, high);
}
```

```
int split(int a[], int low, int high)
{
  int part_element = a[low];

  for (;;) {
    while (low < high && part_element <= a[high])
      high--;
    if (low >= high) break;
    a[low++] = a[high];

    while (low < high && a[low] <= part_element)
      low++;
    if (low >= high) break;
    a[high--] = a[low];
  }

  a[high] = part_element;
  return high;
}
```

Program: Quicksort

- 이 프로그램의 성능을 개선하는 방법:
  - 파티션 알고리즘을 개선.
  - 배열이 크지 않으면 다른 방법을 씀
  - 퀵정렬을 비재귀적으로 만들기