Lecture 9 - Functions

Meng-Hsun Tsai
CSIE, NCKU

Introduction

- A function is a series of statements that have been grouped together and given a name.
- Each function is essentially a small program, with its own declarations and statements.
- Advantages of functions:
  - A program can be divided into small pieces that are easier to understand and modify.
  - We can avoid duplicating code that's used more than once.
  - A function that was originally part of one program can be reused in other programs.

Defining and Calling Functions

- Before we go over the formal rules for defining a function, let’s look at three simple programs that define functions.

Program #1: Computing Averages

- A function named `average` that computes the average of two `double` values:

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

- The word `double` at the beginning is the ***return type*** of `average`.
- The identifiers `a` and `b` (the function's ***parameters***) represent the numbers that will be supplied when `average` is called.

Program #1: Computing Averages (cont.)

- Every function has an executable part, called the ***body***, which is enclosed in braces.
- The body of `average` consists of a single `return` statement.
- Executing this statement causes the function to "return" to the place from which it was called; the value of `(a + b) / 2` will be the value returned by the function.

IMSLab Since 2010

Program #1: Computing Averages (cont.)

- A function call consists of a function name followed by a list of ***arguments.***
  - `average(x, y)` is a call of the `average` function.
- Arguments are used to supply information to a function.
  - The call `average(x, y)` causes the values of `x` and `y` to be copied into the parameters `a` and `b`.
- An argument doesn't have to be a variable; any expression of a compatible type will do.
  - `average(5.1, 8.9)` and `average(x/2, y/3)` are legal.

Program #1: Computing Averages (cont.)

- We'll put the call of `average` in the place where we need to use the return value.
- A statement that prints the average of `x` and `y`:

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

  The return value of `average` isn't saved; the program prints it and then discards it.

- If we had needed the return value later in the program, we could have captured it in a variable:

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

Program #1: Computing Averages (cont.)

- The `average.c` program reads three numbers and uses the `average` function to compute their averages, one pair at a time:

```
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 #1: Computing Averages (cont.)

average.c

```
#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 #2: Printing a Countdown

- To indicate that a function has no return value, we specify that its return type is `void`:

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

- `void` is a type with no values.
- A call of `print_count` appears in a statement by itself:

```
print_count(i);
```

- The `countdown.c` program calls `print_count` 10 times inside a loop.

Program #2: Printing a Countdown (cont.)

countdown.c

```
#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 #3: Printing a Pun (Revisited)

- When a function has no parameters, the word `void` is placed in parentheses after the function's name:

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

- To call a function with no arguments, we write the function's name, followed by parentheses:

```
print_pun();
```

  The parentheses *must* be present.

- The `pun2.c` program tests the `print_pun` function.

Program #3: Printing a Pun (Revisited) (cont.)

pun2.c

```
#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;
}
```

Function Definitions

- General form of a ***function definition:***

  return-type function-name （ *parameters* ）

  {

  declarations

  statements

  }

Function Definitions (cont.)

- The return type of a function is the type of value that the function returns.
- Rules governing the return type:
  - Functions may not return arrays.
  - Specifying that the return type is `void` indicates that the function doesn't return a value.
- If the return type is omitted in C89, the function is presumed to return a value of type `int`.
- In C99, omitting the return type is illegal.

Function Definitions (cont.)

- After the function name comes a list of parameters.
- Each parameter is preceded by a specification of its type; parameters are separated by commas.
- If the function has no parameters, the word `void` should appear between the parentheses.

Function Definitions (cont.)

- The body of a function may include both declarations and statements.
- An alternative version of the `average` function:

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

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

Function Definitions (cont.)

- Variables declared in the body of a function can't be examined or modified by other functions.
- In C89, variable declarations must come first, before all statements in the body of a function.
- In C99, variable declarations and statements can be mixed, as long as each variable is declared prior to the first statement that uses the variable.

Function Definitions (cont.)

- The body of a function whose return type is `void` (a "`void` function") can be empty:

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

- Leaving the body empty may make sense as a temporary step during program development.

Function Calls

- A function call consists of a function name followed by a list of arguments, enclosed in parentheses:

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

- If the parentheses are missing, the function won't be called:

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

This statement is legal but has no effect.

Function Calls (cont.)

- A call of a `void` function is always followed by a semicolon to turn it into a statement:

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

- A call of a non-`void` function produces a value that can be stored in a variable, tested, printed, or used in some other way:

```
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 (cont.)

- The value returned by a non-`void` function can always be discarded if it's not needed:

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

  This call is an example of an expression statement: a statement that evaluates an expression but then discards the result.

Function Calls (cont.)

- Ignoring the return value of `average` is an odd thing to do, but for some functions it makes sense.
- `printf` returns the number of characters that it prints.
- After the following call, `num_chars` will have the value 9:

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

- We'll normally discard `printf`'s return value:

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

Program: Testing Whether a Number Is Prime

- The `prime.c` program tests whether a number is prime:

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

- The program uses a function named `is_prime` that returns `true` if its parameter is a prime number and `false` if it isn't.

- `is_prime` divides its parameter `n` by each of the numbers between 2 and the square root of `n`; if the remainder is ever 0, `n` isn't prime.

Program: Testing Whether a Number Is Prime (cont.)

prime.c

```
#include <stdbool.h>
#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

- Either a declaration or a definition of a function must be present prior to any call of the function.
- A ***function declaration*** provides the compiler with a brief glimpse at a function whose full definition will appear later.
- General form of a function declaration:

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

- The declaration of a function must be consistent with the function's definition.
- Here's the `average.c` program with a declaration of `average` added.

Function Declarations (cont.)

```
#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 (cont.)

- Function declarations of the kind we're discussing are known as ***function prototypes.***
- A function prototype doesn't have to specify the names of the function's parameters, as long as their types are present:

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

Arguments

- In C, arguments are ***passed by value:*** when a function is called, each argument is evaluated and its value assigned to the corresponding parameter.
- Since the parameter contains a copy of the argument’s value, any changes made to the parameter during the execution of the function don’t affect the argument.

Arguments (cont.)

- Consider the following function, which raises a number `x` to a power `n`:

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

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

  return result;
}
```

Arguments (cont.)

- Since `n` is a *copy* of the original exponent, the function can safely modify it, removing the need for `i`:

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

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

  return result;
}
```

Arguments (cont.)

- C's requirement that arguments be passed by value makes it difficult to write certain kinds of functions.
- Suppose that we need a function that will decompose a `double` value into an integer part and a fractional part.
- Since a function can't *return* two numbers, we might try passing a pair of variables to the function and having it modify them:

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

Arguments (cont.)

- A call of the function:

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

- Unfortunately, `i` and `d` won't be affected by the assignments to `int_part` and `frac_part`.

Argument Conversions

- C allows function calls in which the types of the arguments don't match the types of the parameters.
- ***Note that the compiler has encountered a prototype prior to the call.***
- The value of each argument is implicitly converted to the type of the corresponding parameter as if by assignment.
- Example: If an `int` argument is passed to a function that was expecting a `double`, the argument is converted to `double` automatically.

Array Arguments

- When a function parameter is a one-dimensional array, the length of the array can be left unspecified:

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

- C doesn't provide any easy way for a function to determine the length of an array passed to it.
- Instead, we'll have to supply the length—if the function needs it—as an additional argument.

Array Arguments (cont.)

- Example:

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

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

  return sum;
}
```

- Since `sum_array` needs to know the length of `a`, we must supply it as a second argument.

Array Arguments (cont.)

- The prototype for `sum_array` has the following appearance:

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

- As usual, we can omit the parameter names if we wish:

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

Array Arguments (cont.)

- When `sum_array` is called, the first argument will be the name of an array, and the second will be its length:

```
#define LEN 100

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

- Notice that we don't put brackets after an array name when passing it to a function:

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

Array Arguments (cont.)

- A function has no way to check that we've passed it the correct array length.
- Suppose that we've only stored 50 numbers in the `b` array, even though it can hold 100.
- We can sum just the first 50 elements by writing

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

- Be careful not to tell a function that an array argument is *larger* than it really is:

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

`sum_array` will go past the end of the array, causing undefined behavior.

Array Arguments (cont.)

- A function is allowed to change the elements of an array parameter, and the change is reflected in the corresponding argument.

- A function that modifies an array by storing zero into each of its elements:

```
void store_zeros(int a[], int n)
{
  int i;
  for (i = 0; i < n; i++)
    a[i] = 0;
}
```

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

Array Arguments (cont.)

- If a parameter is a multidimensional array, only the length of the first dimension may be omitted.
- If we revise `sum_array` so that `a` is a two-dimensional array, we must specify the number of columns in `a`:

```
#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;
}
```

The `return` Statement

- A non-`void` function must use the `return` statement to specify what value it will return.
- The `return` statement has the form

  `return` *expression* `;`

- The expression is often just a constant or variable:

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

- More complex expressions are possible:

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

The `return` Statement (cont.)

- If the type of the expression in a `return` statement doesn't match the function's return type, the expression will be implicitly converted to the return type.
  - If a function returns an `int`, but the `return` statement contains a `double` expression, the value of the expression is converted to `int`.

The **`return`** Statement (cont.)

- `return` statements may appear in functions whose return type is `void`, provided that no expression is given:

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

- Example:

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

The **return** Statement (cont.)

- A `return` statement may appear at the end of a `void` function:

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

  Using `return` here is unnecessary.

- If a non-`void` function fails to execute a `return` statement, the behavior of the program is undefined if it attempts to use the function's return value.

Program Termination

- Normally, the return type of `main` is `int`:

```
int main(void)
{
   ...
}
```

- Omitting the word `void` in `main`'s parameter list remains legal, but—as a matter of style—it's best to include it.

Program Termination (cont.)

- The value returned by `main` is a status code that can be tested when the program terminates.
- `main` should return 0 if the program terminates normally.
- To indicate abnormal termination, `main` should return a value other than 0.
- It's good practice to make sure that every C program returns a status code.

The `exit` Function

- Executing a `return` statement in `main` is one way to terminate a program.
- Another is calling the `exit` function, which belongs to `<stdlib.h>`.
- The argument passed to `exit` has the same meaning as `main`'s return value: both indicate the program's status at termination.
- To indicate normal termination, we'd pass 0:

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

The `exit` Function (cont.)

- Since 0 is a bit cryptic, C allows us to pass `EXIT_SUCCESS` instead (the effect is the same):

```
exit(EXIT_SUCCESS);
```

- Passing `EXIT_FAILURE` indicates abnormal termination:

```
exit(EXIT_FAILURE);
```

- `EXIT_SUCCESS` and `EXIT_FAILURE` are macros defined in `<stdlib.h>`.
- The values of `EXIT_SUCCESS` and `EXIT_FAILURE` are implementation-defined; typical values are 0 and 1, respectively.

The **`exit`** Function (cont.)

- The statement

  `return` ***expression*;**

  in `main` is equivalent to

  `exit(`*expression*`)`**;**

- The difference between `return` and `exit` is that `exit` causes program termination regardless of which function calls it.
- The `return` statement causes program termination only when it appears in the `main` function.

Recursion

- A function is ***recursive*** if it calls itself.
- The following function computes $n!$ recursively, using the formula $n! = n \times (n - 1)!$:

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

Recursion (cont.)

- To see how recursion works, let's trace the execution of the statement

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

`fact(3)` finds that 3 is not less than or equal to 1, so it calls

`fact(2)`, which finds that 2 is not less than or equal to 1, so it calls

`fact(1)`, which finds that 1 is less than or equal to 1, so it returns 1, causing

`fact(2)` to return $2 \times 1 = 2$, causing

`fact(3)` to return $3 \times 2 = 6$.

Recursion (cont.)

- The following recursive function computes x^n, using the formula $x^n = x \times x^{n-1}$.

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

Recursion (cont.)

- We can condense the `power` function by putting a conditional expression in the `return` statement:

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

- Both `fact` and `power` are careful to test a "termination condition" as soon as they're called.
- All recursive functions need some kind of termination condition in order to prevent infinite recursion.

The Quicksort Algorithm

- Recursion is most helpful for sophisticated algorithms that require a function to call itself two or more times.
- Recursion often arises as a result of an algorithm design technique known as ***divide-and-conquer,*** in which a large problem is divided into smaller pieces that are then tackled by the same algorithm.

The Quicksort Algorithm

- A classic example of divide-and-conquer can be found in the popular ***Quicksort*** algorithm.
- Assume that the array to be sorted is indexed from 1 to n.

Quicksort algorithm

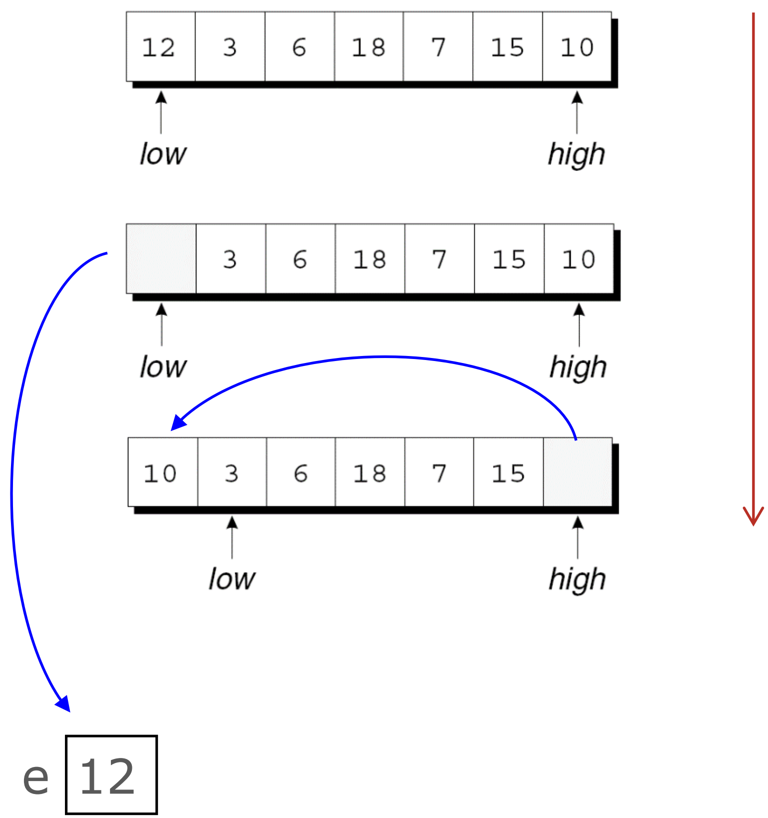
1. Choose an array element e (the "partitioning element"), then rearrange the array so that elements $1, \ldots, i-1$ are less than or equal to e, element i contains e, and elements $i+1, \ldots, n$ are greater than or equal to e.
2. Sort elements $1, \ldots, i-1$ by using Quicksort recursively.
3. Sort elements $i+1, \ldots, n$ by using Quicksort recursively.

The Quicksort Algorithm (cont.)

- Step 1 of the Quicksort algorithm is obviously critical.
- There are various methods to partition an array.
- We'll use a technique that's easy to understand but not particularly efficient.
- The algorithm relies on two "markers" named *low* and *high*, which keep track of positions within the array.

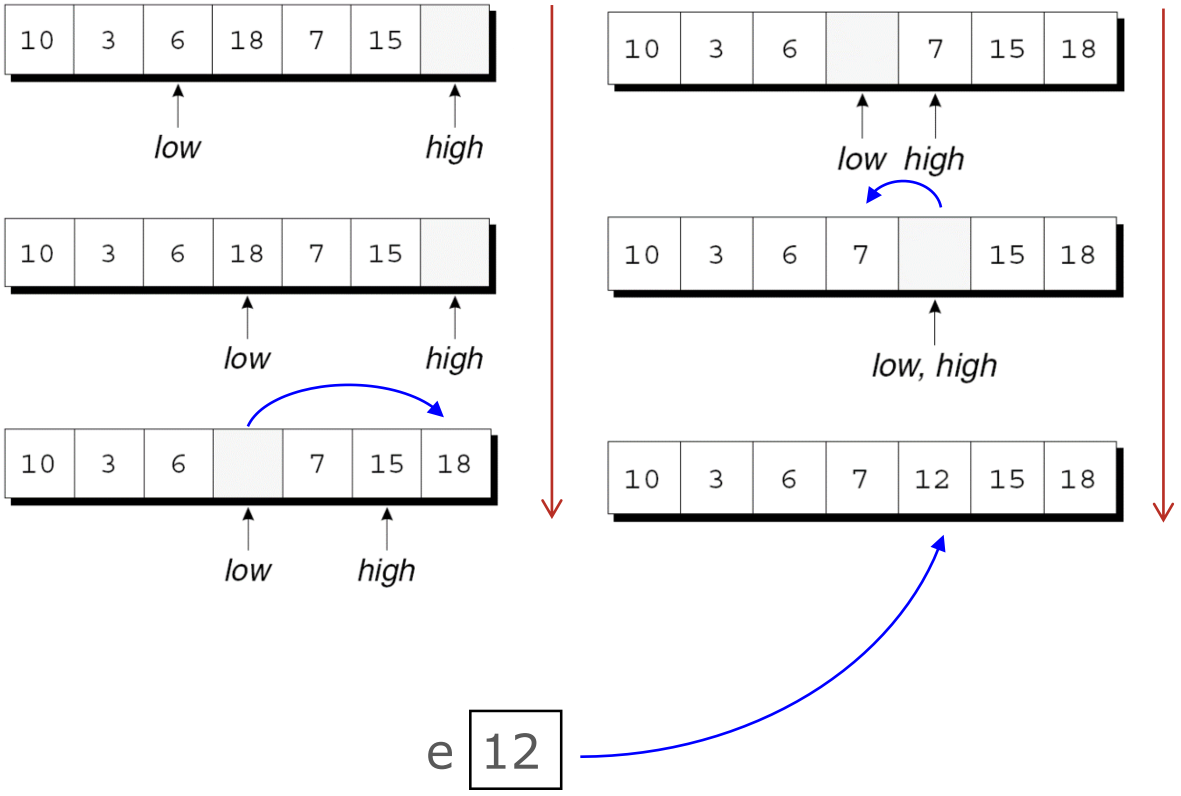
The Quicksort Algorithm (cont.)

- Initially, *low* points to the first element; *high* points to the last.
- We copy the first element (the partitioning element) into a temporary location, leaving a "hole" in the array.
- Next, we move *high* across the array from right to left until it points to an element that's smaller than the partitioning element.
- We then copy the element into the hole that *low* points to, which creates a new hole (pointed to by *high*).



The Quicksort Algorithm (cont.)

- We now move *low* from left to right, looking for an element that's larger than the partitioning element. When we find one, we copy it into the hole that *high* points to.
- The process repeats until *low* and *high* meet at a hole.
- Finally, we copy the partitioning element into the hole.



The Quicksort Algorithm (cont.)

- By the final figure, all elements to the left of the partitioning element are less than or equal to 12, and all elements to the right are greater than or equal to 12.
- Now that the array has been partitioned, we can use Quicksort recursively to sort the first four elements of the array (10, 3, 6, and 7) and the last two (15 and 18).

10	3	6	7	12	15	18

Program: Quicksort (cont.)

- Let's develop a recursive function named `quicksort` that uses the Quicksort algorithm to sort an array of integers.
- The `qsort.c` program reads 10 numbers into an array, calls `quicksort` to sort the array, then prints the elements in the array:

```
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
```

- The code for partitioning the array is in a separate function named `split`.

Program: Quicksort (cont.)

qsort.c

```
#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: ", 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);
}
```

Program: Quicksort (cont.)

```
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 (cont.)

- Ways to improve the program's performance:
  - Improve the partitioning algorithm.
  - Use a different method to sort small arrays.
  - Make Quicksort nonrecursive.

A Quick Review to This Lecture

- General form of a ***function definition:***

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

- Variables declared in the body of a function can't be examined or modified by other functions.
- Function call: function name followed by arguments in parentheses:

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

- This statement is legal but has no effect.
`fun;     // fun()` won't be called

A Quick Review to This Lecture (cont.)

- Specifying return type as `void` indicates no return value.

```
void fun(int n){
}
```

- The word `void` is placed in parentheses indicates that a function has no parameters.

```
void fun(void){
}
fun();
```

The parentheses *must* be present.

- Functions may not return arrays.
- General form of a function declaration:

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

- Either a declaration or a definition of a function must be present prior to any call of the function.

A Quick Review to This Lecture (cont.)

- In C, arguments are ***passed by value:*** when a function is called, each argument is evaluated and its value assigned to the corresponding parameter.
- Passing one-dimensional array, length is supplied as second argument:

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

- Passing two-dimensional array, number of columns must be specified:

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

- The `return` statement has the form

`return` *expression* `;`

A Quick Review to This Lecture (cont.)

- The value returned by `main` is a status code that can be tested when the program terminates. (0: normal, non-0: abnormal)
- Program termination
  - `return` statement in `main()`
  - Calling `exit()` in any function
- A function is ***recursive*** if it calls itself.
- Recursion arises ***divide-and-conquer*** technique: a large problem is divided into smaller pieces that are tackled by the same algorithm.
- All recursive functions need some kind of termination condition in order to prevent infinite recursion.

IMSLab Since 2010