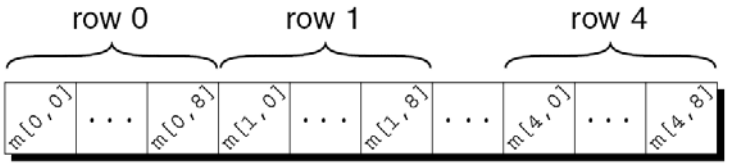
Multidimensional Arrays

- The following declaration creates a two-dimensional array (a *matrix*, in mathematical terminology):

```
int m[5][9];
```

  Array `m` has 5 rows and 9 columns. Both rows and columns are indexed from 0:

- To access the element of m in row i, column j, we must write `m[i][j]`, not `m[i,j]`.
- C stores arrays in ***row-major order,*** with row 0 first, then row 1, and so forth.
- The m array is stored in the memory as follow:



- We can create an initializer for a two-dimensional array by nesting one-dimensional initializers:

```
int m[5][9] = {{1,1,1,1,1,0,1,1,1},
               {0,1,0,1,0,1,0,1,0},
               {0,1,0,1,1,0,0,1,0},
               {1,1,0,1,0,0,0,1,0},
               {1,1,0,1,0,0,1,1,1}};
```

- If an initializer isn't large enough to fill a multidimensional array, the remaining elements are given the value 0.

  The following initializer fills only the first three rows of m; the last two rows will contain zeros:

```
int m[5][9] = {{1,1,1,1,1,0,1,1,1},
               {0,1,0,1,0,1,0,1,0},
               {0,1,0,1,1,0,0,1,0}};
```

- If an inner list isn't long enough to fill a row, the remaining elements in the row are initialized to 0:

```
int m[5][9] = {{1,1,1,1,1,0,1,1,1},
               {0,1,0,1,0,1,0,1},
               {0,1,0,1,1,0,0,1},
               {1,1,0,1,0,0,0,1},
               {1,1,0,1,0,0,1,1,1}};
```

- We can even omit the inner braces:

```
int m[5][9] = {1,1,1,1,1,0,1,1,1,
               0,1,0,1,0,1,0,1,0,
               0,1,0,1,1,0,0,1,0,
               1,1,0,1,0,0,0,1,0,
               1,1,0,1,0,0,1,1,1};
```

  Once the compiler has seen enough values to fill one row, it begins filling the next.

- Omitting the inner braces can be risky, since an extra element or a missing element will affect the rest of the initializer.
- C99's designated initializers work with multidimensional arrays.

  Example:

```
double a[2][2]={[0][0]=1.0,[1][1]=1.0};
```

  As usual, all elements for which no value is specified will default to zero.

Constant Arrays

- An array can be made "constant" by starting its declaration with the word const:

```
const char hex_chars[] =
{'0', '1', '2', '3', '4', '5', '6',
 'A', 'B', 'C', 'D', 'E', 'F'};
```

- An array that's been declared const should not be modified by the program.
- Advantages of declaring an array to be const:
  - Documents that the program won't change the array.
  - Helps the compiler catch errors.
- `const` isn't limited to arrays.

Variable-Length Arrays (C99)

- In C89, the length of an array variable must be specified by a constant expression.
- In C99, however, it's sometimes possible to use an expression that's *not* constant.

  Example:

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

void main(void) {
  int i, n;

  printf("Pleae enter n:  ");
  scanf("%d", &n);

  int a[n]; /* C99 only */
  printf("Enter %d numbers: ", n);
  scanf("%d", &n) ;
  for (i = 0; i < n; i++)
    scanf("%d", &a[i]);
}
```

- The length of a VLA doesn't have to be specified by a single variable. Arbitrary expressions are legal:

```
int a[3*i+5], b[j+k];
```

- Like other arrays, VLAs can be multidimensional:

```
int c[m][n];
```

- Restrictions on VLAs:
  – Can't have static storage.
  – Can't have an initializer.

Chapter 9 Functions

- 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

- A simple example of defining a function :

  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.

- 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.

- An example of calling a function.

```
#include <stdio.h>
double average(double a, double b) {
       return (a + b) / 2;
}
void main(void) {
  double x=1.2, y=2.3, z;
  z = average(x,y) ;
  printf("Average= %g\n", z);
}
```

- 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, y/3)` are legal.

Example 2:

```
#include <stdio.h>
double avg(double a, double b) {
    double c ;
    c = a + b;
    return c/2 ;
}
void main(void) {
  double x=1.2, y=2.3, z;
  z = avg(x,y) ;
  printf("Average= %g\n", z);
}
```

Example 3:

```
#include <stdio.h>
void avg(double a, double b, double *c){
    double c ;
   *c = (a + b) / 2;
}
void main(void) {
  double x=1.2, y=2.3, z;
  average(x,y. &c) ;
  printf("Average= %g\n", z);
}
```

Function Definitions

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

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

- 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.
- 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.
- The body of a function may include both declarations and statements.

  Example 2:

```
double average(double a, double b) {
   double c;       /* declaration */
   c = a + b;      /* statement */
   return c / 2;   /* statement */
}
```

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

```
void function1( ... )
{}
```

- 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);

- 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("average=%g\n", average(x, y));
```

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

  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 has the value 5:

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

  We'll normally discard printf's return value:

```
printf("Hi! \n");
```

- To make it clear that we're deliberately discarding the return value of a function, C allows us to put (void) before the call:

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

  Using (void) makes it clear to others that you deliberately discarded the return value, not just forgot that there was one.

Function Declarations

- C doesn't require that the definition of a function precede its calls.

  But we have to declare each function before calling it.

- 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.

Example

```
#include <stdio.h>
double average(double a, double b);
/* DECLARATION */
int main(void) {
  double x, y, z;
  printf("Enter two numbers: ");
  scanf("%lf%lf", &x, &y);
  printf("Average of %g and %g: %g\n",
         x, y, average(x, y));
  return 0;
}

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

- Function declarations 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);
```

- It's usually best not to omit parameter names.
- C99 has adopted the rule that either a declaration or a definition of a function must be present prior to any call of the function.

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.
- The fact that arguments are passed by value has both advantages and disadvantages.
- Since a parameter can be modified without affecting the corresponding argument, we can use parameters as variables within the function, reducing the number of genuine variables needed.
- 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;
}
```

- 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;
}
```

- **Pass by value** makes it difficult to write certain functions.

  Suppose that we need a function that will decompose a double value into an integer part and a fractional part. We want a function to produce two numbers as output:

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

  There is no way to pass two numbers as output using pass by value.

Argument Conversions

- C allows function calls in which the types of the arguments don't match the types of the parameters.

  The rules governing how the arguments are converted depend on whether or not the compiler has seen a prototype for the function (or the function's full definition) prior to the call.

- ***If the compiler has encountered a prototype prior to the call,*** then the value of each argument is implicitly converted to the type of the corresponding parameter as if by assignment.
- ***If the compiler has not encountered a prototype prior to the call,*** then the compiler performs the ***default argument promotions:***
  - float arguments are converted to double.
  - The integral promotions are performed, causing char and short arguments to be converted to int.

- Example:

```
#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;
}
```

- At the time square is called, the compiler doesn't know that it expects an argument of type int. Relying on the default argument promotions is dangerous.
- In C99, calling square without first providing a declaration or definition of the function is an error.

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

int f(int []);
```

- 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.

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

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

- A function has no way to check that we've passed it the correct array length.
- A function is allowed to change the elements of an array parameter, and the change is reflected in the corresponding argument., meaning can pass the computing result back to the calling function.

  Example:

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

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

- If a parameter is a multidimensional array, only the length of the first dimension may be omitted.

  Example:

```
#define LEN 10

int sum_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;
}
```

- Not being able to pass multidimensional arrays with an arbitrary number of columns can be a nuisance.

  We can often work around this difficulty by using arrays of pointers.

  C99's **variable-length array** parameters provide another solution.

- Consider the sum_array function:

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

- Using a variable-length array parameter, we can explicitly state that a's length is n:

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

  Note that the order of the parameters has been switched; order is important when variable-length array parameters are used.

- The function can also have the prototypes:

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

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

- In general, the length of a variable-length array parameter can be any expression.

```
int function(int m, int n,
             int a[m], int b[n],
             int c[m+n]) ;
```

Compound Literals (C99).

- In C99, we can use a ***compound literal:*** an unnamed array that's created "on the fly" by simply specifying which elements it contains:

Example:

```
int sum_array(int a[], int n) {
   . . .
}
main() {
  int b[] = []){3,0,3,1};
     . . .
  total=sum_array((int []){3,0,3,1},4);
     . . .
}
```

- We also have the option of specifying a length explicitly:

  ```
  (int [4]){ 3,0,3,1}
  ```

  is equivalent to

  ```
  (int []){3,0,3,1}
  ```

- A compound literal resembles a cast applied to an initializer.
- In fact, compound literals and initializers obey the same rules.
- A compound literal may contain designators, just like a designated initializer. If it does not provide full initialization, any uninitialized elements default to zero.

  For example, the literal (int [10]){8, 6} has 10 elements; the first two have the values 8 and 6, and the remaining elements have the value 0.
- Compound literals created inside a function may contain arbitrary expressions, not just constants:

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

- If desired, a compound literal can be made "read-only" by adding the word const to its type:

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

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. More complex expressions are possible:

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

- If the type of the expression in a return statement doesn't match function's return type, the expression will be implicitly converted to the return type.

  For example, if a function returns an int, but the return statement contains a double expression, the value of the expression is converted to int.

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

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

  More examples:

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

void print_pun(void)
   printf("To C, or not to C.\n");
   return;  // OK, but not needed.
}
```

- 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.
- Normally, the return type of main is int.
- Omitting the word void in main's parameter list remains legal, but—as a matter of style—it's best to include it.
- Omitting the return type of a function isn't legal in C99, so it's best to avoid this practice.
- 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
exit(EXIT_SUCCESS);
```

- 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 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);
}
```

- How recursion works. Let's look at fact(3).

  fact(3) will 3 * fact(2),

  and fact(2) returns 2*fact(1).

  fact(1) returns 1. So, fact(3) to return $3 \times 2 = 6$.
- We can condense the fact function by putting a conditional expression in the return statement:

```
int fact(int n) {
   return ( (n==1)?1:(n*fact(n-1)) );
}
```

- All recursive functions need some kind of termination condition in order to prevent infinite recursion.