

## 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; typically are 0 and 1.
- 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  $3 * \text{fact}(2)$ ,  
and fact(2) returns  $2 * \text{fact}(1)$ .  
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)) );  
}
```

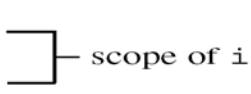
- All recursive functions need some kind of termination condition in order to prevent infinite recursion.

## Chp 10. Program Organization

### Local Variables

- A variable declared in the body of a function is said to be *local* to the function.
- **Automatic storage duration.** The storage for a local variable is “automatically” allocated when the enclosing function is called and deallocated when the function returns, unless the variable is declared static.
- **Block scope.** A local variable is visible from its point of declaration to the end of the enclosing function body.
- Since C99 doesn’t require variable declarations to come at the beginning of a function, it’s possible for a local variable to have a very small scope. For example,

```
void f(void)
{
    ...
    int i;
    ...
}
```



### Static Local Variables

- Including static in the declaration of a local variable causes it to have *static storage duration*.  
A variable with static storage duration has a permanent storage location, so it retains its value throughout the execution of the program.
- A static local variable still has block scope, so it’s not visible to other functions.
- Example:

```
void f(void) {
    static int i; //static local variable
    . . .
}
```

### Parameters

- Parameters have the same properties—automatic storage duration and block scope—as local variables.
- Each parameter is initialized automatically when a function is called (by being assigned the value of the corresponding argument).

### External Variables

- Passing arguments is one way to transmit information to a function.
- Functions can also communicate through *external variables*—variables that are declared outside the body of any function.
- External variables are sometimes known as *global variables*.
- Properties of external variables:
  - Static storage duration
  - File scope
- Having *file scope* means that an external variable is visible from its point of declaration to the end of the enclosing file.
- Example:

```
#include <stdio.h>

int contents[100];
int top = 0;

void function1(void){
    top = 0;
}

. . .
```

### Pros and Cons of External Variables

- External variables are convenient when many functions must share a variable or when a few functions share a large number of variables.
- In most cases, it’s better for functions to communicate through parameters rather than by sharing variables:
  - If we change an external variable during program maintenance (by altering its type, say), we’ll need to check every function in the same file to see how the change affects it.
  - If an external variable is assigned an incorrect value, it may be difficult to identify the guilty function.
  - Functions that rely on external variables are hard to reuse in other programs.
- Don’t use the same external variable for different purposes in different functions.

Example: Suppose that several functions need a variable named *i* to control a for statement. Instead of declaring *i* in each function that uses it, some programmers declare it just once at the top of the program. This practice is misleading; someone reading the program later may think that the uses of *i* are related, when in fact they’re not.

- Make sure that external variables have meaningful names.
- Local variables don't always need meaningful names: it's often hard to think of a better name than `i` for the control variable in a `for` loop.
- Making variables external when they should be local can lead to some rather frustrating bugs.

Example:

```
int i;

void print_one_row(void) {
    for(i = 1; i <= 10; i++)
        printf("*");
}

void print_all_rows(void) {
    for (i = 1; i <= 10; i++) {
        print_one_row();
        printf("\n");
    }
}
```

- Instead of printing 10 rows, `print_all_rows` prints only one.

## Blocks

- In Section 5.2, we encountered compound statements of the form
 

```
{ statements }
```
- C allows compound statements to contain declarations as well as statements:

```
{ declarations statements }
```

- This kind of compound statement is called a **block**.

Example of a block:

```
if(i>j) { // swap values of i and j
    int temp = i;
    i = j;
    j = temp;
}
```

- By default, the storage duration of a variable declared in a block is automatic: storage for the variable is allocated when the block is entered and deallocated when the block is exited.

The variable has block scope; it can't be referenced outside the block.

- A variable that belongs to a block can be declared static to give it static storage duration.

- Blocks are useful inside a function body when we need variables for temporary use.
- Advantages of declaring temporary variables in blocks:
  - Avoids cluttering declarations at the beginning of the function body with variables that are used only briefly.
  - Reduces name conflicts.
- C99 allows variables to be declared anywhere within a block.

## Scope

- In a C program, the same identifier may have different meanings.

A scope rule: When a declaration inside a block names an identifier that's already visible, the new declaration temporarily "hides" the old one, and the identifier takes on a new meaning.

At the end of the block, the identifier regains its old meaning.

```
int i; /* Declaration 1 */

void f(int i) /* Declaration 2 */
{
    i = 1;
}

void g(void)
{
    int i = 2; /* Declaration 3 */
    if (i > 0) {
        int i; /* Declaration 4 */
        i = 3;
    }
    i = 4;
}

void h(void)
{
    i = 5;
}
```

- In the example, C's scope rules allow us to determine the meaning of `i` each time it's used (indicated by arrows).

## C Program Organization

- Major elements of a C program:
  - Preprocessing directives such as `#include` and `#define`
  - Type definitions
  - Declarations of external variables
  - Function prototypes
  - Function definitions
- C imposes a few rules on the order of these items:
  - A preprocessing directive doesn't take effect until the line on which it appears.
  - A type name can't be used until it's been defined.
  - A variable can't be used until it's declared.
- It's a good idea to define or declare every function prior to its first call. C99 makes this a requirement.
- There are several ways to organize a program so that these rules are obeyed.
  - One possible ordering:
    - `#include` directives
    - `#define` directives
    - Type definitions
    - Declarations of external variables
    - Prototypes for functions other than `main`
    - Definition of `main`
    - Definitions of other functions
- It's a good idea to have a **boxed comment** preceding each function definition.
  - Information to include in the comment:
    - Name of the function
    - Purpose of the function
    - Meaning of each parameter
    - Description of return value (if any)
    - Description of side effects (such as modifying external variables)