Chapter 9
Statements

A statement is a complete instruction to the computer. Except as indicated, statements are executed in sequence. Statements have the form:

statement:
  expression−statement
  compound−statement
  selection−statement
  iteration−statement
  jump−statement
  labeled−statement

Expression Statement

Most statements are expression statements, which have the form:

expression−statement:
  $expression_{opt};$

Usually expression statements are expressions evaluated for their side effects, such as assignments or function calls. A special case is the *null statement,* which consists of only a semicolon.

Compound Statement or Block

A compound statement (or block) groups a set of statements into a syntactic unit. The set can have its own declarations and initializers, and has the form:

compound−statement:
  $\{declaration{-}list_{opt}\ statement{-}list_{opt}\}$

declaration−list:
  declaration
  declaration−list declaration

statement−list:
  statement
  statement−list statement

Declarations within compound statements have *block scope*. If any of the identifiers in the *declaration−list* were previously declared, the outer declaration is hidden for the duration of the block, after which it resumes its force. In traditional C, however, function declarations always have *file scope* whenever they appear.

Initialization of identifiers declared within the block is restricted to those that have no linkage. Thus, the initialization of an identifier declared within the block using the **extern** specifier is not allowed. These initializations are performed only once, prior to the first entry into the block, for identifiers with static storage duration. For identifiers with automatic storage duration, it is performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case, no initializations are performed.

Selection Statements

Selection statements include **if** and **switch** statements and have the form:

selection–statement:
  if (expression) statement
  if (expression) statement else statement
  switch (expression) statement

Selection statements choose one of a set of statements to execute, based on the evaluation of the expression. The expression is referred to as the controlling expression.

The *if* Statement

The controlling expression of an **if** statement must have scalar type.

For both forms of the **if** statement, the first statement is executed if the controlling expression evaluates to nonzero. For the second form, the second statement is executed if the controlling expression evaluates to zero. An **else** clause that follows multiple sequential **else**–less**if** statements is associated with the most recent **if** statement in the same block (that is, not in an enclosed block).

The *switch* Statement

The controlling expression of a **switch** statement must have integral type. The statement is typically a compound statement, some of whose constituent statements are labeled **case** statements (see "Labeled Statements"). In addition, at most one labeled **default** statement can occur in a **switch**. The expression on each **case** label must be an integral constant expression. No two expressions on **case** labels in the same switch can evaluate to the same constant.

A compound statement attached to a **switch** can include declarations. Due to the flow of control in a **switch**, however, initialization of identifiers so declared are not performed if these initializers have automatic storage duration.

The integral promotions are performed on the controlling expression, and the constant expression of each **case** statement is converted to the promoted type. Control is transferred to the labeled **case** statement whose expression value matches the value of the controlling expression. If no such match occurs, control is transferred either past the end of the **switch** or to the labeled **default** statement, if one exists in the **switch**. Execution continues sequentially once control has been transferred. In particular, the flow of control is not altered upon encountering another **case** label. The **switch** statement is exited, however, upon encountering a **break** or **continue** statement (see "The break Statement" and "The continue Statement", respectively).

A simple example of a complete **switch** statement is:

```
switch (c) {
 case 'o':
```

```
 case ’o’:
  oflag = TRUE;
  break;
 case ’p’:
  pflag = TRUE;
  break;
 case ’r’:
  rflag = TRUE;
  break;
 default :
  (void) fprintf(stderr,
          "Unknown option\n");
  exit(2);
}
```

Iteration Statements

Iteration statements execute the attached statement (called the *body*) repeatedly until the controlling expression evaluates to zero. In the **for** statement, the second expression is the controlling expression. The format is:

iteration−statement:
while (expression) statement
do statement while (expression) ;
for (expression$_{opt}$ *; expression*$_{opt}$ *; expression*$_{opt}$*) statement*

The controlling expression must have scalar type.

The flow of control in an iteration statement can be altered by a jump−statement (see"Jump Statements").

The *while* Statement

The controlling expression of a **while** statement is evaluated before each execution of the body.

The *do* Statement

The controlling expression of a **do** statement is evaluated after each execution of the body.

The *for* Statement

The **for** statement has the form:

for (expression$_{opt}$ *; expression*$_{opt}$ *; expression*$_{opt}$*)*
statement

The first expression specifies initialization for the loop. The second expression is the controlling expression, which is evaluated before each iteration. The third expression often specifies incrementation. It is evaluated *after* each iteration.

This statement is equivalent to:

expression−1;
while (expression−2)
{
statement
expression−3;
}

One exception exists, however. If a **continue** statement (see "The continue Statement") is encountered, *expression−3* of the **for** statement is executed prior to the next iteration.

Any or all of the expressions can be omitted. A missing expression−2 makes the implied **while** clause equivalent to *while* (1). Other missing expressions are simply dropped from the expansion above.

Jump Statements

Jump statements cause unconditional transfer of control. The syntax is:

jump−statement:
goto identifier;
continue;
break;
return expression $_{opt}$*;*

The *goto* Statement

Control can be transferred unconditionally by means of a **goto** statement:

goto identifier;

The identifier must name a label located in the enclosing function. If the label has not yet appeared, it is implicitly declared. (See "Labeled Statements" for more information.)

The *continue* Statement

The continue statement can appear only in the body of an iteration statement. It causes control to pass to the loop−continuation portion of the smallest enclosing **while**, **do**, or **for** statement—that is, to the end of the loop. More precisely, consider each of the following statements:

```
while (...)
{
..
contin: ;
}

do {
   ...
   contin: ;
} while (...) ;
```

```
for (...) {
    ...
    contin: ;
}
```

A continue is equivalent to `goto contin.` Following the `contin:` is a null statement.

The *break* Statement

The break statement can appear only in the body of an iteration statement or code attached to a switch statement. It transfers control to the statement immediately following the smallest enclosing iteration or switch statement, terminating its execution.

The *return* Statement

A function returns to its caller by means of the return statement. The value of the expression is returned to the caller after conversion, as if by assignment, to the declared type of the function, as the value of the function call expression. The return statement cannot have an expression if the type of the current function is void.

If the end of a function is reached prior to the execution of an explicit return, an implicit return (with no expression) is executed. If the value of the function call expression is used when none is returned, the behavior is undefined.

Labeled Statements

Labeled statements have the following syntax:

labeled−statement:
    identifier : statement
    case constant−expression : statement
    default : statement

A case or default label can appear only on statements that are part of a **switch.**

Any statement can have a label attached as a simple identifier. The scope of such a label is the current function. Thus, labels must be unique within a function. In traditional C, identifiers used as labels and in object declarations share a name space. Thus, use of an identifier as a label hides any declaration of that identifier in an enclosing scope. In ANSI C, identifiers used as labels are placed in a different name space from all other identifiers, and do not conflict. Thus the following code fragment is legal in ANSI C, but not in traditional C.

```
{
    int foo;
    foo = 1;
    …
    goto foo;
```

```
    …
    foo: ;
}
```