

Conditional Operator

- The operator: `expr1 ? expr2 : expr3`
- The operands can be of any type.
- The resulting expression is a **conditional expression**.
- The expression is evaluated in stages:
 - `expr1` is evaluated first; if its value isn't zero, then `expr2` is evaluated, and its value is the value of the entire conditional expression.
 - If the value of `expr1` is zero, then the value of `expr3` is the value of the conditional.

Example:

```
int i, j, k;

i = 1;
j = 2;
k = (i > j ? i : j);           // k is now 2
k = ((i >= 0 ? i : 0) + j);    // k is now 3
...
printf("%d\n", i > j ? i : j);

return (i > j ? i : j) ;
```

Boolean Values

- C89 lacks a proper Boolean type. One way to work around this limitation is to declare an `int` variable and then assign it either 0 or 1:

```
int flag;
flag = 0;
...
flag = 1;
```

- Although this scheme works, it doesn't contribute much to program readability.
- To make programs more understandable, C89 programmers often define macros with names such as `TRUE` and `FALSE`.

```
#define TRUE      1
#define FALSE     0
```

```
flag = FALSE;
...
flag = TRUE;
```

- C99 provides the `_Bool` type. `_Bool` is an integer type, so a `_Bool` variable is really just an integer variable in disguise. A Boolean variable can be declared by writing
`_Bool flag;`
- Unlike an ordinary integer variable, however, a `_Bool` variable can only be assigned 0 or 1.
- Attempting to store a nonzero value into a `_Bool` variable will cause the variable to be assigned 1:
`flag = 5; /* flag is assigned 1 */`
- It's legal (although not advisable) to perform arithmetic on `_Bool` variables.
- It's also legal to print a `_Bool` variable (either 0 or 1 will be displayed).
- C99's `<stdbool.h>` header makes it easier to work with Boolean values.
- It defines a macro, `bool`, that stands for `_Bool`.
- If `<stdbool.h>` is included, we can write
`bool flag; /* same as _Bool flag; */`
- `<stdbool.h>` also supplies macros named `true` and `false`, which stand for 1 and 0, respectively, making it possible to write
`flag = false;`
...
`flag = true;`

The switch Statement

```
switch ( expression ) {
    case constant-expression : statements
        ...
    case constant-expression : statements
    default : statements
}
```

- `switch` must be followed by an integer expression—the *controlling expression*—in parentheses.
- Characters are treated as integers in C and thus can be tested in `switch` statements.
- Floating-point numbers and strings don't qualify, however.
- A **constant expression** can't contain variables or function calls.

Code segment

```
if (grade == 4)
    printf("Excellent");
else if (grade == 3)
    printf("Good");
else if (grade == 2)
    printf("Average");
else if (grade == 1)
    printf("Poor");
else if (grade == 0)
    printf("Failing");
else
    printf("Illegal grade");
```

can be replaced by

```
switch (grade) {
    case 4: printf("Excellent");
           break;
    case 3: printf("Good");
           break;
    case 2: printf("Average");
           break;
    case 1: printf("Poor");
           break;
    case 0: printf("Failing");
           break;
    default: printf("Illegal grade");
            break;
}
```

- After each case label comes any number of statements.
No braces are required around the statements.
- The last statement in each group is normally break.
Without break (or some other jump statement) at the end of a case, control will flow into the next case.

Example:

```
switch (grade) {
    case 4: printf("Excellent");
    case 3: printf("Good");
    case 2: printf("Average");
    case 1: printf("Poor");
    case 0: printf("Failing");
    default: printf("Illegal grade");
}
```

If the value of grade is 3, the message printed is GoodAveragePoorFailingIllegal grade

- Duplicate case labels aren't allowed.
- The order of the cases doesn't matter, and the default case doesn't need to come last.
- If the default case is missing and the controlling expression's value doesn't match any case label, control passes to the next statement after the switch.
- Several case labels can be put on the same line:

```
switch (grade) {
    case 4:
    case 3:
    case 2:
    case 1: printf("Passing");
           break;
    case 0: printf("Failing");
           break;
    default: printf("Illegal grade");
            break;
}
```

```
switch (grade) {
    case 4: case 3: case 2: case 1:
           printf("Passing");
           break;
    case 0: printf("Failing");
           break;
    default: printf("Illegal grade");
            break;
}
```

Chp 6 Loops

- A **loop** is a statement whose job is to repeatedly execute some other statement (the **loop body**).
- Every loop has a **controlling expression**.
- Each time the loop body is executed (an **iteration** of the loop), the controlling expression is evaluated.
 - If the expression is true (has a value that's not zero) the loop continues to execute.
- Three iteration statements:
 - while(expression) { statement; }
 - do {statement;} while(expression) ;
 - for (expr1 ; expr2 ; expr3) {statement;}

The while Statement

```
while(controlling expression) {  
    statements          // the loop body  
}
```

- The controlling expression is evaluated first.
- If the controlling expression is evaluated to be nonzero, the loop body is executed, and then the execution returns to evaluate the controlling expression.
- If the controlling expression is evaluated to be zero, the loop body is not executed and the execution goes to execute the line after the while statement.
- Example:

```
i = 1;  
n = 10 ;  
while (i < n) {  
    i = i * 2 ;  
}
```

- Observations about the while statement:
 - The controlling expression is false when a while loop terminates.
 - The body of a while loop may not be executed at all, because the controlling expression is tested *before* the body is executed.
 - A while statement won't terminate if the controlling expression always has a nonzero value.
 - The controlling expression can be called the loop termination condition. Sometimes, people may choose to write the loop-termination condition inside the loop. For example,

```
while(1) {  
    , , ,  
  
    if(expression1) break ;  
    , , ,  
  
    if(expression2) break ;  
    , , ,  
  
}
```
- Besides break, there are other statements (go to, return) can also break out of the loop.

The do-while Statement

```
do {  
    statements // loop body  
while ( expression ) ;
```

- The loop body is executed first, then the expression is evaluated.
- If the value of the expression is nonzero, the loop body is executed again and then the expression is evaluated once more.
- If the value of the expression is zero, the loop body is not executed.

The for Statement

```
for(expr1; expr2; expr3) {  
    statements  
}
```

- *expr1* is an initialization step that's performed only once, before the loop begins to execute.
- *expr2* controls loop termination (the loop continues executing as long as the value of *expr2* is nonzero).
- *expr3* is an operation to be performed at the end of each loop iteration.
- Equivalent to the code

```
expr1;  
while ( expr2 ) {  
    statements;  
    expr3;  
}
```

- Example: Re-write the code using for statement

```
i = 1;  
n = 10 ;  
while (i < n) {  
    printf("i = %d\n", i*2) ;  
}
```

Solution:

```
n = 10 ;  
for(i=1; i<n; i++) {  
    printf("i = %d\n", i*2) ;  
}
```

- C allows any or all of the expressions that control a for statement to be omitted.
- If the *first* expression is omitted, no initialization is performed before the loop is executed. For example,

```
i = 1;
for( ; i < 10; i++) {
    printf("i = %d\n", i*2) ;
}
```

- If the *second* expression is missing, it defaults to a true value, so the for statement doesn't terminate (unless stopped in some other fashion)..
- If the *third* expression is omitted, the loop body is responsible for ensuring that the value of the second expression eventually becomes false. For example,

```
for(i=1; i<10; ) {
    printf("i = %d\n", i*2) ;
    i++ ;
}
```

- When the *first* and *third* expressions are both omitted, the resulting loop is nothing more than a while statement in disguise:

```
for( ; i > 0; ) {
    statement;
    i--;
}
```

is the same as

```
while (i > 0) {
    statement;
    i--;
}
```

- The while version is clearer and therefore preferable.

- In **C99**, the first expression in a for statement can be replaced by a declaration.
- This feature allows the programmer to declare a variable for use by the loop:

```
for (int i = 0; i < n; i++)
    ...
```

- The variable *i* need not have been declared prior to this statement.

- A variable declared by a for statement can't be accessed outside the body of the loop (we say that it's not *visible* outside the loop):

```
for (int i = 0; i < n; i++) {
    ...
    printf("%d", i); // legal. i is visible inside loop
    ...
}
printf("%d", i); // WRONG
```

- A for statement may declare more than one variable, provided that all variables have the same type:

```
for (int i = 0, j = 0; i < n; i++)
```

- A **comma expression** as the first or third expression in the for statement.
- A comma expression has the form
expr1 , *expr2*
where *expr1* and *expr2* are any two expressions.

- Example:

```
for (sum = 0, i = 1; i <= N; i++)
    sum += i;
```

- A comma expression is evaluated in two steps:
 - First, *expr1* is evaluated and its value discarded.
 - Second, *expr2* is evaluated; its value is the value of the entire expression.
- Evaluating *expr1* should always have a side effect; if it doesn't, then *expr1* serves no purpose.
- When the comma expression ++i, i + j is evaluated, i is first incremented, then i + j is evaluated.

Exiting from a Loop

- The normal exit point for a loop is at
 - the beginning (as in a while or for statement), or
 - the end (the do while statement).
- The **break** statement: make it possible to
 - exit in the middle of the loop, or
 - provide a loop with more than one exit point.
- The break statement can transfer control out of a switch statement, but it can also be used to jump out of a while, do while, or for loop.

Example:

```
while (1) {
    printf("Enter a number (enter 0 to stop): ");
    scanf("%d", &n);
    if (n == 0)
        break;
    printf("%d cubed is %d\n", n, n * n * n);
}
```

- A break statement transfers control out of the innermost enclosing while, do, for, or switch.
- When these statements are nested, the break statement can escape only one level of nesting.
- Example:

```
while (...) {
    switch (...) {
        ...
        break;
        ...
    }
}
```

break transfers control out of the switch statement, but not out of the while loop.

- The **continue** statement is similar to break:
 - break transfers control just past the end of a loop. With break, control leaves the loop.
 - continue transfers control to a point just before the end of the loop body. With continue, control remains inside the loop
- There's another difference between break and continue: break can be used in switch statements and loops (while, do, and for), whereas continue is limited to loops.

```
n = 0;
sum = 0;
while (n < 10) {
    scanf("%d", &i);
    if(i == 0) continue;
    sum += i;
    n++;
    /* continue jumps to here */
}
```

is equivalent to

```
n = 0;
sum = 0;
while (n < 10) {
    scanf("%d", &i);
    if(i != 0) {
        sum += i;
        n++;
    }
}
```

The goto Statement

- The goto statement is capable of jumping to any statement in a function, provided that the statement has a **label**.
- A label is just an identifier placed at the beginning of a statement:
identifier : statement
- A statement may have more than one label.
- The goto statement itself has the form
goto identifier ;
- Executing the statement *goto L*; transfers control to the statement that follows the label *L*, which must be in the same function as the goto statement itself.

Example:

```
for(d = 2; d < n; d++) {
    if(n % d == 0) goto done;
}
done:
if(d < n)
    printf("%d is divisible by %d\n", n, d);
else
    printf("%d is prime\n", n);
```

The Null Statement

- A statement devoid of symbols except for the semicolon at the end.
- To avoid confusion, C programmers customarily put the null statement on a line by itself.

```
i = 10;
while(i > 0);
{
    printf("i is %d\n", i);
    --i;
}
```

- Accidentally putting a semicolon after the parentheses in an if, while, or for statement creates a null statement.