Flow of Control

Chapter 4
### Operator precedence and Associativity

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ) ++(postfix) --(postfix)</td>
<td>left to right</td>
</tr>
<tr>
<td>+(unary) -(unary) ++(prefix) --(prefix) !</td>
<td>right to left</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>right to left</td>
</tr>
<tr>
<td>= += -= *= /= etc</td>
<td>right to left</td>
</tr>
<tr>
<td>, (coma operator)</td>
<td>left to right</td>
</tr>
</tbody>
</table>
The if Statement

if (expr)
  statement

For example

if ( grade >= 90 )
  printf("congratulations\n" );
  printf(" your grade is %d.\n",grade );

Statement may be a compound statement
if ( j < k ){
  min = j;
  printf(" j is smaller than k\n");
}

The *if-else* Statement

```c
if (expr)
    statement1
else
    statement2
```

For example

```c
if (x < y)
    min = x;
else
    min = y;
```
The if–else Statement

- An if statement can be embedded within another if statement.
- Indentation is irrelevant. An else always attaches to the nearest if. So

```c
if (a == 1)
    if (b == 2)
        printf("***\n");
else
    printf("###\n");
```

is equivalent to

```c
if (a == 1) {
    if (b == 2)
        printf("***\n");
} else
    printf("###\n");
```

/* not equivalent */
The **while** Statement

```
while (expr)
    statement
next statement
```

For example:
```
while ( i++ < n )
    factorial *= i;

while ( ( c = getchar() ) != EOF ){
    if ( c >= 'a' && c <= 'z' )
        ++lowercase_letter_cnt;
    ++total_cnt;
}
```
/ Count blanks, digits, letters, newlines, and others. */
#include <stdio.h>

int main(void)
{
    int blank_cnt = 0, c, digit_cnt = 0,
        letter_cnt = 0, nl_cnt = 0, other_cnt = 0;

    while ((c = getchar()) != EOF) /* braces not necessary */
        if (c == ' ')
            ++blank_cnt;
        else if (c >= '0' && c <= '9')
            ++digit_cnt;
        else if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
            ++letter_cnt;
        else if (c == '\n')
            ++nl_cnt;
        else
            ++other_cnt;

    printf("%10s%10s%10s%10s%10s\n", 
        "blanks", "digits", "letters", "lines", "others", "total");

    printf("%10d%10d%10d%10d%10d\n", 
        blank_cnt, digit_cnt, letter_cnt, nl_cnt, other_cnt,
        blank_cnt + digit_cnt + letter_cnt + nl_cnt + other_cnt);

    return 0;
}
The **for** Statement

```c
for ( expr1; expr2; expr3 )
  statement
next statement
```

is equivalent to

```c
expr1;
while ( expr2 ) {
  statement
  expr3;
}
next statement
```
The **for** Statement - examples

```
int factorial = 1;
for ( i = 1; i <= 10; ++i )
    factorial *= i;

or

for ( j = 2; k % j == 0; ++j ) {
    printf("%d is a divisor of %d\n", j, k);
    sum += j;
}
```
The **for** Statement

Any or all expressions in the for statement may be missing:
The interesting case: if expr2 is missing – it is evaluated as true.

```c
i = 1;
sum = 0;
for ( ; i <= 10; ++i)
  sum += i;
```

What is wrong with the following code?

```c
i = 1;
sum = 0;
for ( ; ; ) {
  for ( ; ; ) {
    sum += i++;
    printf("%d\n", sum);
  }
}
The **for** Statement

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

is equivalent to:

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

but is not equivalent to:

```c
for ( sum = 0, i = 1; i <= n; ++i, sum += i );
```
Print a table of values for some Boolean functions

#include <stdio.h>
int main( void )
{
  int b1 = 0, b2 = 0, b3 = 0, b4 = 0, b5 = 0;
  int cnt = 0;

  /* headings */
  printf( "\n%5s%5s%5s%5s%5s%5s%7s%7s%11s\n\n",
    "Cnt", "b1", "b2", "b3", "b4", "b5", "fct1", "fct2", "majority" );
Print a table of values for some Boolean functions

```c
for ( b1 = 0; b1 <= 1; ++b1 )
    for ( b2 = 0; b2 <= 1; ++b2 )
        for ( b3 = 0; b3 <= 1; ++b3)
            for ( b4 = 0; b4 <= 1; ++b4)
                for ( b5 = 0; b5 <= 1; ++b5)
                    printf("%5d%5d%5d%5d%5d%5d%6d%7d%9d\n", ++cnt, b1, b2, b3, b4, b5,
                        b1 || b3 || b5, b1 && b2 || b4 && b5,
                        b1 + b2 + b3 + b4 + b5 >= 3 );

    putchar(\n');
return 0;
```
The comma Operator

In a comma expression of the form

\[ expr1, \ expr2 \]

the \( expr1 \) is evaluated first, then \( expr2 \). The comma expression has value and type of its right operand. For example

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i = 1, \ j = 2, \ ++k + 1 )</td>
<td>( (( \ i = 1 ), \ ( j = 2 ), \ (( \ ++k \ ) + 1 ) )</td>
<td>5</td>
</tr>
<tr>
<td>( k != 1, \ ++x * 2.0 + 1 )</td>
<td>( ( k != 1 ), \ ((( \ ++x ) * 2.0 ) + 1 ) )</td>
<td>9.6</td>
</tr>
</tbody>
</table>
The do Statement

do
  statement
while (expr)

For example:

do {
  printf("Input a positive integer: ");
  scanf("%d", &n);
  if (error = (n <= 0))
    printf("\nERROR; Do it again!\n\n" );
} while (error);
/* a test that fails. File loop.c */
#include <stdio.h>
int main( void )
{
    int  cnt = 0;
    double sum = 0.0, x = 0.0;

    for ( x = 0.0; x != 9.9; x += 0.1 )
    {
        sum += x;
        printf( "cnt = %5d\n", ++cnt );
    }
    printf( "sum = %f\n", sum );
    return 0;
}
Example: Fibonacci Numbers

/* Print Fibonacci Numbers and Quotients */
#include <stdio.h>
#define LIMIT 46
int main(void)
{
    long f0 = 0, f1 = 1, n = 0, temp = 0;

    printf("%7s\19s\n%7s\19s\n%7s\19s\n", " ", "Fibonacci", "Fibonacci", " n", " number", "quotient",
        "--", "--------", "--------");
    printf("%7d\19d\n%7d\19d\n", 0, 0, 1, 1); /* first 2 cases */
    for ( n = 2; n <= LIMIT; ++n ) {
        temp = f1;
        f1 += f0;
        f0 = temp;
        printf("%7ld%19ld%29.16f\n", n, f1, (double) f1 / f0);
    }
    return 0;
}
<table>
<thead>
<tr>
<th>n</th>
<th>Fibonacci number</th>
<th>Fibonacci quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1.0000000000000000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2.0000000000000000</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1.5000000000000000</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1.6666666666666667</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1.6000000000000000</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>1.6250000000000000</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>1.61803399001755971</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td>1.6180339887498949</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>28657</td>
<td>1.6180339887498949</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>701408733</td>
<td>1.6180339887498949</td>
</tr>
<tr>
<td>46</td>
<td>1836311903</td>
<td>1.6180339887498949</td>
</tr>
</tbody>
</table>
The goto Statement
This statement is considered harmful in most programming methodologies.
It causes an unconditional jump to a labeled statement.
The scope of the label is within the function it occurs.
Multiple labels on a single statement are allowed.

goto error;
.....
error: {
    printf(" An error occurred -- bye!\n " );
    exit(1);
}

The **break** Command

The two special statements **break** and **continue** interrupt the normal flow of control;

**break** causes an exit from the enclosing loop or a **switch** command.

For example:

```c
while ( 1 ) {
    scanf("%1f", &x );
    if ( x < 0.0 )
        break;
    printf("%f\n", sqrt( x ) );
}
```
The continue Command

continue may occur only inside for, while and do loops. It transfers control to the end of the current iteration, without terminating the loop. For example:

```c
for ( i = 0; i < TOTAL; ++i ) {
    c = getchar();
    if ( c >= `0' && c <= `9' )
        continue;
    ...
    /* process other character types */
}
/* continue transfers control to here */
```
The Switch Statement

The `switch` statement is a multiway generalization of the `if-else` statement. For example:

```c
switch ( c ) {
    case 'a':
        ++a_cnt;
        break;
    case 'b':
        ++b_cnt;
        break;
    case 'c':
    case 'C':
        ++c_cnt;
        break;
    default:
        ++other_cnt;
}
```

Important `switch` rules:
- The controlling expression must be of an integral type (in the example it is the int variable `c`.
- The constants in the `case` labels must all be unique.
- Usually, each case ends with a `break`, otherwise execution continues into the next case – a frequent source of error.
The conditional operator

The conditional operator `?` is a ternary operator with the following construct

```
expr1 ? expr2: expr3
```

`expr1` is evaluated first. If it is nonzero (true) then `expr2` is evaluated giving its value to the whole expression. Otherwise, `expr3` is evaluated and is the value of the expression.

```
char a = 'a', b = 'b'; /* a is numerically 97 */
int i = 1, j = 2;
double x = 7.07;
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i == j ? a - 1 : b + 1</code></td>
<td>99</td>
<td>int</td>
</tr>
<tr>
<td><code>j % 3 == 0 ? i + 4 : x</code></td>
<td>7.07</td>
<td>double</td>
</tr>
<tr>
<td><code>j % 3 ? i + 4 : x</code></td>
<td>5.0</td>
<td>double</td>
</tr>
</tbody>
</table>