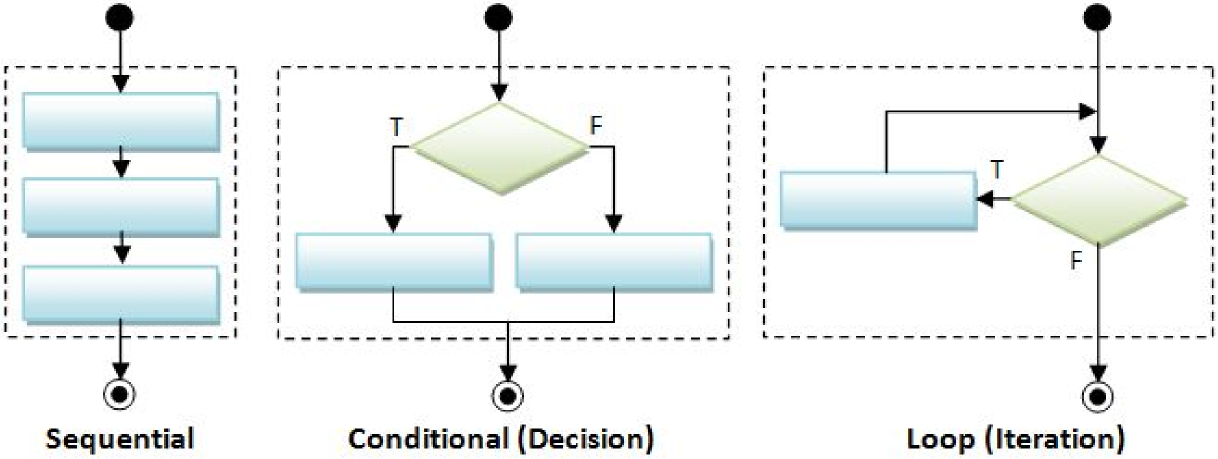
03. Flow Control

There are three basic flow control constructs - *sequential*, *conditional* (or *decision*), and *loop* (or *iteration*), as illustrated below.



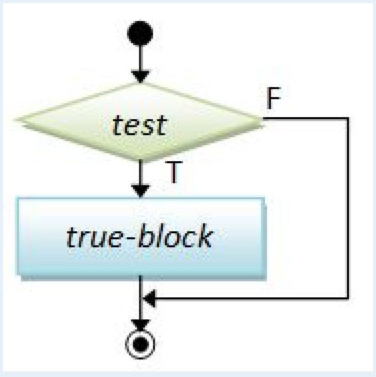
03.1 Sequential Flow Control

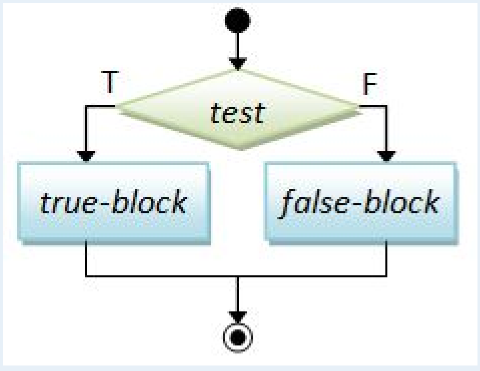
A program is a sequence of instructions. *Sequential* flow is the most common and straight-forward, where programming statements are executed in the order that they are written - from top to bottom in a sequential manner.

03.2 Conditional (Decision) Flow Control

There are a few types of conditionals, *if-then*, *if-then-else*, *nested-if* (*if-elseif-elseif-...-else*), *switch-case*, and *conditional expression*.

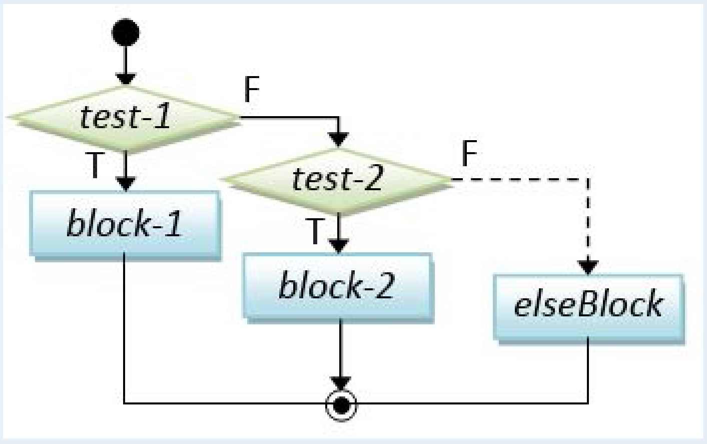
Syntax	Example	Flowchart

`// if-then`
`if` `( `*`booleanExpression`*` ) {`
    `true-block`` ;`
`}`
	`if (mark >= 50) {`
	    `printf("Congratulation!\n");`
	    `printf("Keep it up!\n");`
	`}`
		

`// if-then-else`
`if` `( `*`booleanExpression`*` ) {`
    `true-block`` ;`
`} `**`else`**` {`
    `false-block`` ;`
`}`
	`if (mark >= 50) {`
	    `printf("Congratulation!\n");`
	    `printf("Keep it up!\n");`
	`} else {`
	    `printf("Try Harder!\n");`
	`}`
		

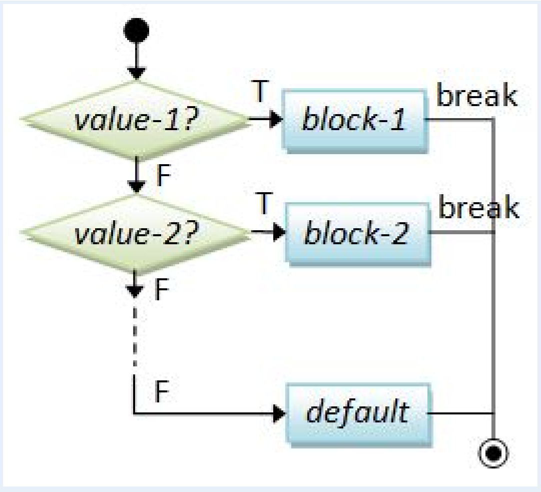
```
// nested-if
if ( booleanExpr-1 ) {
   block-1 ;
} else if ( booleanExpr-2 ) {
   block-2 ;
} else if ( booleanExpr-3 ) {
   block-3 ;
} else if ( booleanExpr-4 ) {
   ......
} else {
   elseBlock ;
}
```

```
if (mark >= 80) {
   printf("A\n");
} else if (mark >= 70) {
   printf("B\n");
} else if (mark >= 60) {
   printf("C\n");
} else if (mark >= 50) {
   printf("D\n");
} else {
   printf("F\n");
}
```



```
// switch-case
switch ( selector ) {
   case value-1:
      block-1; break;
   case value-2:
      block-2; break;
   case value-3:
      block-3; break;
   ......
   case value-n:
      block-n; break;
   default:
      default-block;
}
```

```
char oper; int num1, num2, result;
......
switch (oper) {
   case '+':
      result = num1 + num2; break;
   case '-':
      result = num1 - num2; break;
   case '*':
      result = num1 * num2; break;
   case '/':
      result = num1 / num2; break;
   default:
      printf("Unknown operator\n");
}
```



"switch-case" is an alternative to the "nested-if". In a *switch-case* statement, a `break` statement is needed for each of the cases. If `break` is missing, execution will flow through the following case. You can use either an `int` or `char`variable as the case-*selector*.

Conditional Operator: A conditional operator is a ternary (3-operand) operator, in the form of *`booleanExpr ? trueExpr : falseExpr`*. Depending on the *`booleanExpr`*, it evaluates and returns the value of *`trueExpr`* or *`falseExpr`*.

Syntax	Example

`booleanExpr ? trueExpr : falseExpr`	`printf("%s\n", (mark >= 50) ? "PASS" : "FAIL");`
	`   // print either "PASS" or "FAIL"`
	`max = (a > b) ? a : b;   // RHS returns a or b`
	`abs = (a > 0) ? a : -a;  // RHS returns a or -a`

Braces: You could omit the braces { }, if there is only one statement inside the block. For example,

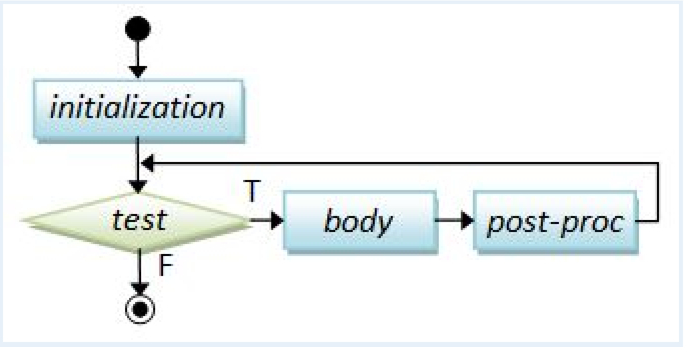
```
if (mark >= 50)
   printf("PASS\n");    // Only one statement, can omit { } but not recommended
else {                  // more than one statements, need { }
   printf("FAIL\n");
   printf("Try Harder!\n");
}
```

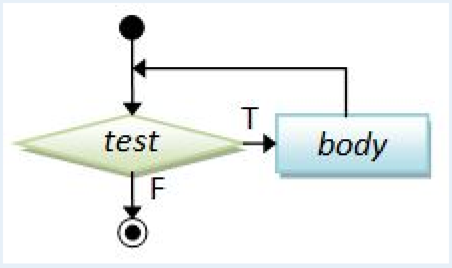
However, I recommend that you keep the braces, even though there is only one statement in the block, to improve the readability of your program.

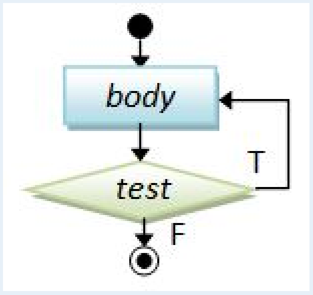
03.3 Loop Flow Control

Again, there are a few types of loops: *for-loop*, *while-do*, and *do-while*.

Syntax	Example	Flowchart

`// for-loop`
`for (init; test; post-proc) {`
`    body ;`
`}`
	`// Sum from 1 to 1000`
	`int sum = 0, number;`
	`for (number = 1; number <= 1000; ++number) {`
	`    sum += number;`
	`}`
		

`// while-do`
`while ( condition ) {`
`    body ;`
`}`
	`int sum = 0, number = 1;`
	`while (number <= 1000) {`
	`    sum += number;`
	`    ++number;`
	`}`
		

`// do-while`
`do {`
`    body ;`
`}`
`while ( condition ) ;`
	`int sum = 0, number = 1;`
	`do {`
	`    sum += number;`
	`    ++number;`
	`} while (number <= 1000);`
		

The difference between *while-do* and *do-while* lies in the order of the *body* and *condition*. In *while-do*, the *condition* is tested first. The body will be executed if the *condition* is true and the process repeats. In *do-while*, the *body* is executed and then the *condition* is tested. Take note that the *body* of *do-while* will be executed at least once (vs. possibly zero for *while-do*).

Suppose that your program prompts user for a number between 1 to 10, and checks for valid input, do-while with a boolean flag could be more appropriate.

```
// Input with validity check
bool valid = false;
int number;
do {
  // prompt user to enter an int between 1 and 10
  ......
  // if the number entered is valid, set done to exit the loop
  if (number >=1 && number <= 10) {
     valid = true;
  }
} while (!valid);   // Need a semi-colon to terminate do-while
```

Below is an example of using while-do:

```
// Game loop
bool gameOver = false;
while (!gameOver) {
   // play the game
   ......
   // Update the game state
   // Set gameOver to true if appropriate to exit the game loop
   ......
}
```

Example (Counter-Controlled Loop): Prompt user for an upperbound. Sum the integers from 1 to a given upperbound and compute its average.

```c
/*
 * Sum from 1 to a given upperbound and compute their average (SumNumbers.c)
 */
#include <stdio.h>

int main() {
   int sum = 0;    // Store the accumulated sum
   int upperbound;

   printf("Enter the upperbound: ");
   scanf("%d", &upperbound);

  // Sum from 1 to the upperbound
   int number;
   for (number = 1; number <= upperbound; ++number) {
      sum += number;
   }
   printf("Sum is %d\n", sum);
   printf("Average is %.2lf\n", (double)sum / upperbound);

  // Sum only the odd numbers
   int count = 0;    // counts of odd numbers
   sum = 0;          // reset sum
   for (number = 1; number <= upperbound; number = number + 2) {
      ++count;
      sum += number;
   }
   printf("Sum of odd numbers is %d\n", sum);
   printf("Average is %.2lf\n", (double)sum / count);
}
```

Example (Sentinel-Controlled Loop): Prompt user for positive integers, and display the count, maximum, minimum and average. Terminate when user enters -1.

```
/* Prompt user for positive integers and display the count, maximum,
   minimum and average. Terminate the input with -1 (StatNumbers.c) */
#include <stdio.h>
#include <limits.h> // for INT_MAX

int main() {
   int numberIn = 0; // input number (positive integer)
   int count = 0;    // count of inputs, init to 0
   int sum = 0;      // sum of inputs, init to 0
   int max = 0;      // max of inputs, init to minimum
   int min = INT_MAX; // min of inputs, init to maximum (need <climits>)
   int sentinel = -1; // Input terminating value

  // Read Inputs until sentinel encountered
   printf("Enter a positive integer or %d to exit: ", sentinel);
   scanf("%d", &numberIn);
   while (numberIn != sentinel) {
     // Check input for positive integer
      if (numberIn > 0) {
         ++count;
         sum += numberIn;
         if (max < numberIn) max = numberIn;
         if (min > numberIn) min = numberIn;
```

```
        } else {
            printf("error: input must be positive! try again...\n");
        }
        printf("Enter a positive integer or %d to exit: ", sentinel);
        scanf("%d", &numberIn);
    }

    // Print result
    printf("\n");
    printf("Count is %d\n", count);
    if (count > 0) {
        printf("Maximum is %d\n", max);
        printf("Minimum is %d\n", min);
        printf("Average is %.2lf\n", (double)sum / count);
    }
}
```

Program Notes

- In computing, a *sentinel value* is a special value that indicates the end of data (e.g., a negative value to end a sequence of positive value, end-of-file, null character in the null-terminated string). In this example, we use -1 as the sentinel value to indicate the end of inputs, which is a sequence of positive integers. Instead of hardcoding the value of -1, we use a variable called `sentinel` for flexibility and ease-of-maintenance.
- Take note of the *while-loop pattern* in reading the inputs. In this pattern, you need to *repeat* the prompting and input statement.

03.4 Interrupting Loop Flow - "break" and "continue"

The break statement breaks out and exits the current (innermost) loop.

The continue statement aborts the current iteration and continue to the next iteration of the current (innermost) loop.

break and continue are poor structures as they are hard to read and hard to follow. Use them only if absolutely necessary. You can always write the same program without using break and continue.

Example (break): The following program lists the non-prime numbers between 2 and an upperbound.

```c
/*
 *  List non-prime from 1 to an upperbound (NonPrimeList.c).
 */
#include <stdio.h>
#include <math.h>

int main() {
   int upperbound, number, maxFactor, factor;
   printf("Enter the upperbound: ");
   scanf("%d", &upperbound);
   for (number = 2; number <= upperbound; ++number) {
     // Not a prime, if there is a factor between 2 and sqrt(number)
      maxFactor = (int)sqrt(number);
      for (factor = 2; factor <= maxFactor; ++factor) {
         if (number % factor == 0) {  // Factor?
            printf("%d ", number);
            break;  // A factor found, no need to search for more factors
         }
      }
   }
   printf("\n");
   return 0;
}
```

Let's rewrite the above program without using break statement. A while loop is used (which is controlled by the boolean flag) instead of for loop with break.

```c
/*
 *  List primes from 1 to an upperbound (PrimeList.c).
 */
#include <stdio.h>
#include <math.h>

int main() {
   int upperbound, number, maxFactor, isPrime, factor;
   printf("Enter the upperbound: ");
   scanf("%d", &upperbound);

   for (number = 2; number <= upperbound; ++number) {
     // Not prime, if there is a factor between 2 and sqrt of number
      maxFactor = (int)sqrt(number);
      isPrime = 1;
      factor = 2;
      while (isPrime && factor <= maxFactor) {
         if (number % factor == 0) {  // Factor of number?
             isPrime = 0;
         }
         ++factor;
      }
      if (isPrime) printf("%d ", number);
   }
   printf("\n");
   return 0;
}
```

Example (continue):

```c
// Sum 1 to upperbound, exclude 11, 22, 33,...
int upperbound = 100;
int sum = 0;
int number;
for (number = 1; number <= upperbound; ++number) {
   if (number % 11 == 0) continue;  // Skip the rest of the loop body, continue to the next iteration
   sum += number;
}
// It is better to re-write the loop as:
for (number = 1; number <= upperbound; ++number) {
   if (number % 11 != 0) sum += number;
}
```

Example (`break` and `continue`): Study the following program.

```
/* A mystery series (Mystery.c) */
#include <stdio.h>

int main() {
   int number = 1;
   while (1) {
      ++number;
      if ((number % 3) == 0) continue;
      if (number == 133) break;
      if ((number % 2) == 0) {
         number += 3;
      } else {
         number -= 3;
      }
      printf("%d ", number);
   }
   printf("\n");
   return 0;
}
```

03.5 Terminating Program

There are a few ways that you can terminate your program, before reaching the end of the programming statements.

`exit()`: You could invoke the function `exit(int exitCode)`, in `<stdlib.h>`, to terminate the program and return the control to the Operating System. By convention, return code of zero indicates normal termination; while a non-zero `exitCode` (-1) indicates *abnormal termination*. For example,

`abort()`: The header `<stdlib.h>` also provide a function called `abort()`, which can be used to terminate the program *abnormally*.

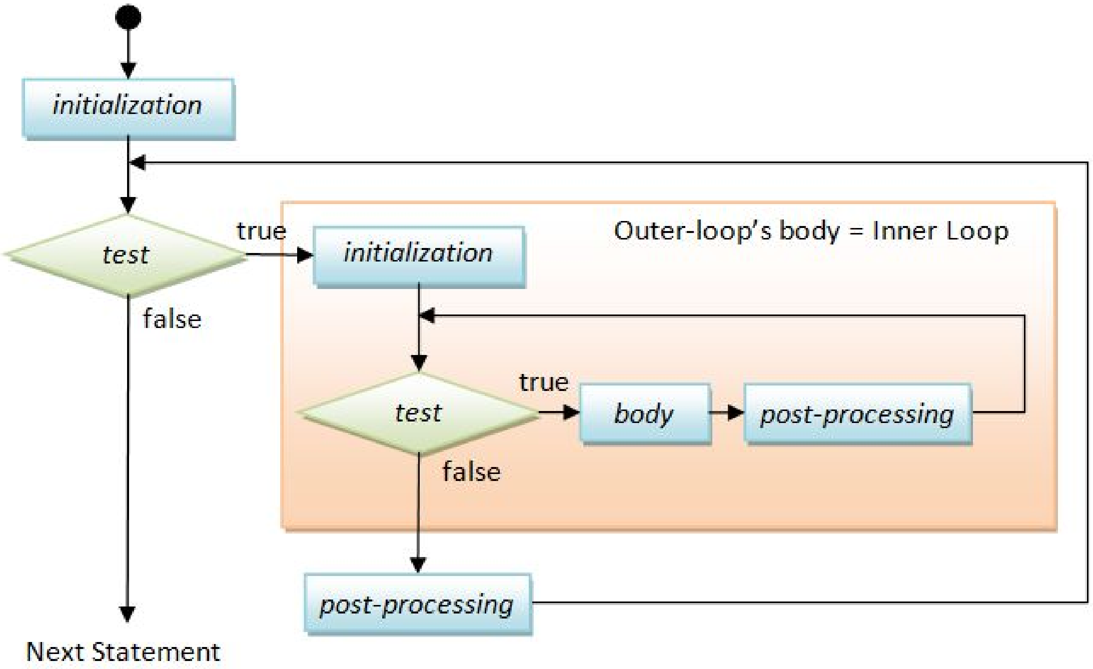
```
if (errorCount > 10) {
   printf("too many errors\n");
   exit(-1);  // Terminate the program
              // OR abort();
}
```

The "`return`" Statement: You could also use a "`return` *`returnValue`*" statement in the `main()` function to terminate the program and return control back to the Operating System. For example,

```
int main() {
   ...
   if (errorCount > 10) {
      printf("too many errors\n");
      return -1;  // Terminate and return control to OS from main()
   }
   ...
}
```

03.6 Nested Loops

The following diagram illustrates a nested for-loop, i.e., an inner for-loop within an outer for-loop.



Try out the following program, which prints a 8-by-8 checker box pattern using *nested loops*, as follows:

```
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
```

```
/*
 *  Print square pattern (PrintSquarePattern.c).
 */
#include <stdio.h>

int main() {
   int size = 8, row, col;
   for (row = 1; row <= size; ++row) {     // Outer loop to print all the rows
      for (col = 1; col <= size; ++col) { // Inner loop to print all the columns of
each row
         printf("# ");
      }
      printf("\n");  // A row ended, bring the cursor to the next line
   }

   return 0;
 }
```

This program contains two *nested* for-loops. The inner loop is used to print a row of eight "# ", which is followed by printing a newline. The outer loop repeats the inner loop to print all the rows.

Suppose that you want to print this pattern instead (in program called `PrintCheckerPattern.c`):

```
# # # # # # # #
 # # # # # # # #
# # # # # # # #
 # # # # # # # #
# # # # # # # #
 # # # # # # # #
# # # # # # # #
 # # # # # # # #
```

You need to print an additional space for even-number rows. You could do so by adding the following statement before Line 8.

```
if ((row % 2) == 0) {   // print a leading space for even-numbered rows
   printf(" ");
}
```

Exercises

1. Print these patterns using nested loop (in a program called `PrintPattern1x`). Use a variable called `size` for the size of the pattern and try out various sizes. You should use as few `printf()` statements as possible.

```
 # * # * # * # *    # # # # # # # #    # # # # # # # #    1                                1
# * # * # * # *     # # # # # # #        # # # # # # #    2 1                            1 2
 # * # * # * # *    # # # # # #            # # # # # #    3 2 1                        1 2 3
# * # * # * # *     # # # # #                # # # # #    4 3 2 1                    1 2 3 4
 # * # * # * # *    # # # #                    # # # #    5 4 3 2 1                1 2 3 4 5
# * # * # * # *     # # #                        # # #    6 5 4 3 2 1            1 2 3 4 5 6
 # * # * # * # *    # #                            # #    7 6 5 4 3 2 1        1 2 3 4 5 6 7
# * # * # * # *     #                                #    8 7 6 5 4 3 2 1    1 2 3 4 5 6 7 8
     (a)                  (b)                (c)                (d)                (e)
```

Hints: The equations for major and opposite diagonals are row = col and row + col = size + 1. Decide on what to print above and below the diagonal.

2. Print the timetable of 1 to 9, as follows, using nested loop. (Hints: you need to use an *if-else* statement to check whether the product is single-digit or double-digit, and print an additional space if needed.)

```
1  2  3  4  5  6  7  8  9
2  4  6  8 10 12 14 16 18
 ......
```

3. Print these patterns using nested loop.

```
# # # # # # #      # # # # # # #      # # # # # # #      # # # # # # #      # # # # # # #
#           #        #                          #          #       #        # #       # #
#           #          #                      #              #   #          #   #   #   #
#           #            #                  #                  #            #     #     #
#           #              #              #                  #   #          #   #   #   #
#           #                #          #                  #       #        # #       # #
# # # # # # #      # # # # # # #      # # # # # # #      # # # # # # #      # # # # # # #
     (a)                 (b)               (c)                (d)                (e)
```

03.7 Some Issues in Flow Control

Dangling `else`: The "dangling else" problem can be illustrated as follows:

```
if (i == 0)
   if (j == 0)
      printf("i and j are zero\n");
else printf("i is not zero\n");   // intend for the outer-if
```

The `else` clause in the above codes is syntactically applicable to both the outer-if and the inner-if. The C compiler always associate the `else` clause with the innermost if (i.e., the nearest if). Dangling else can be resolved by applying explicit parentheses. The above codes are logically incorrect and require explicit parentheses as shown below.

```
if ( i == 0) {
   if (j == 0) printf("i and j are zero\n");
} else {
   printf("i is not zero\n");   // non-ambiguous for outer-if
}
```

Endless Loop: The following constructs:

```
while (1) { ...... }
```

is commonly used. It seems to be an endless loop (or infinite loop), but it is usually terminated via a `break` or `return` statement inside the loop body. This kind of code is hard to read - avoid if possible by re-writing the condition.