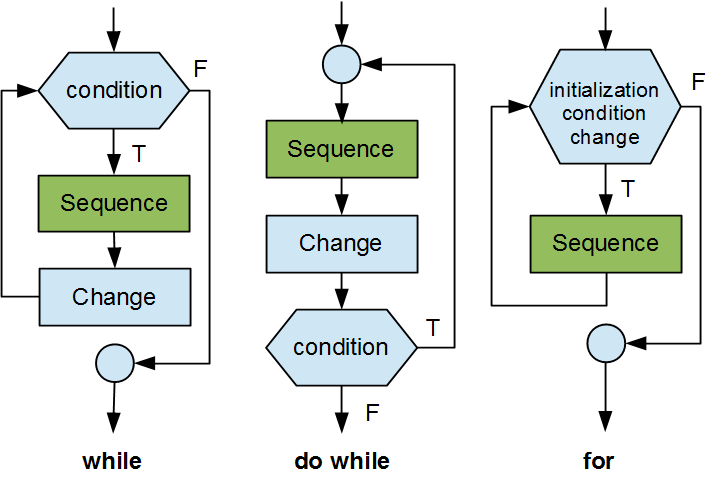
**Iteration Constructs**

The C language supports three iteration constructs:

* while
* do while
* for

Three instructions control the execution of an iteration: an initialization, a test condition, a change statement.  The test condition may be simple or compound.  The flow charts for the three constructs are shown below.



If the change statement is missing or if the test condition is always satisfied, the iteration continues without terminating and the program can never terminate.  We say that such an iteration is an *infinite loop*.

while

The while construct executes its sequence as long as the test condition is true.  This construct takes the form

while (*condition*)

*sequence*

For the code snippet below, the output is shown on the right

|  |  |
| --- | --- |
| slices = 4;  while (slices > 0) {  slices--;  printf("Gulp! Slices left %d\n", slices);  } | Gulp! Slices left 3  Gulp! Slices left 2  Gulp! Slices left 1  Gulp! Slices left 0 |

If the condition is never true (for example, if initially slice = 0), this construct never executes the sequence.

do while

The do while construct executes its sequence at least once and continues executing it as long as the test condition is true.  This construct takes the form

do

*sequence*

while (*condition*);

For the code snippet below, the output is shown on the right

|  |  |
| --- | --- |
| slices = 4;  do {  slices--;  printf("Gulp! Slices left %d\n", slices);  } while (slices > 0); | Gulp! Slices left 3  Gulp! Slices left 2  Gulp! Slices left 1  Gulp! Slices left 0 |

If we change the initial value to slices = 12 and the test condition to slices < 5, this iteration displays once and stops because the test condition is false.

|  |  |
| --- | --- |
| slices = 12;  do {  slices--;  printf("Gulp! Slices left %d\n", slices);  } while (slices < 5); | Gulp! Slices left 11 |

This code probably contains a semantic error: if the initial value was 5, the iteration would never end.

for

The for construct groups the initialization, test condition and change together, separating them with semi-colons.  This construct takes the form

for (*initialization*; *condition*; *change*)

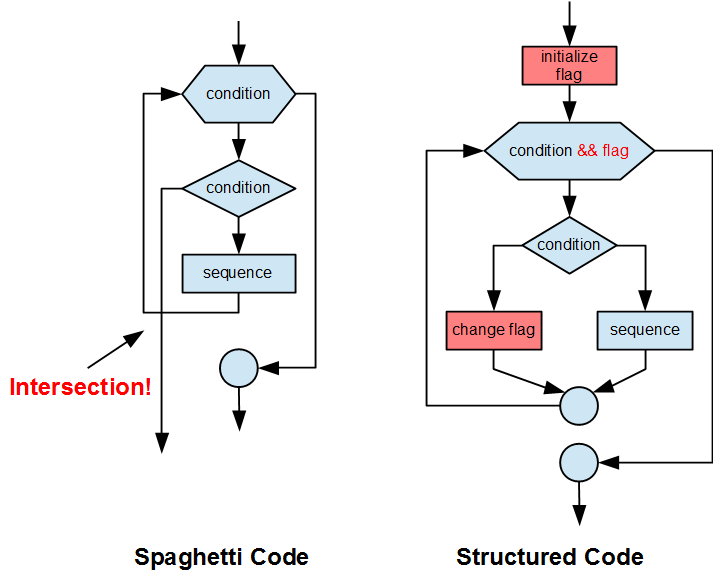
*sequence*

For the code snippet below, the output is shown on the right

|  |  |
| --- | --- |
| for (slices = 4; slices > 0; --slices)  printf("Gulp! Slices left %d\n", slices - 1); | Gulp! Slices left 3  Gulp! Slices left 2  Gulp! Slices left 1  Gulp! Slices left 0 |

Flags

Flagging is a method of coding iteration constructs within the single-entry single-exit rule of structured programming.  Consider the flow-chart on the left side in the figure below.  This design contains a path that crosses another path.



*Flags* are variables that determine whether an iteration continues or stops.  A flag is either true or false.  Flags helps ensure that no paths cross one another.  By introducing a flag, we avoid the jump and multiple exit, obtain a flow chart where no path crosses any other and hence an improved design.

Example

The following code snippet uses a flag to terminate the iteration prematurely.  The input/output is shown on the right:

|  |  |
| --- | --- |
| int done = 0; // flag  int total = 0; // accumulator  for (i = 0; i < 10 && done == 0; i++) {  printf("Enter integer (0 to stop) ");  scanf("%d", &value);  if (value == 0)  done = 1;  else  total += value;  }  printf("Total = %d\n", total); | Enter integer (0 to stop) 45  Enter integer (0 to stop) 32  Enter integer (0 to stop) 3  Enter integer (0 to stop) -6  Enter integer (0 to stop) 0  Total = 74 |

The test condition is compound due to the evaluation of the flag.  If done == 1, the iteration stops.

Avoid Jumps

Designing a program with jumps or intersecting paths makes it more difficult to read.  We refer to program code that contains paths that cross one another as *spaghetti code*.  The roots of spaghetti coding lie in assembly languages (second-generation languages).  Assembly languages include jump instructions.  Jump instructions migrated to high-level languages as assembly language programmers started coding in higher-level languages.  Spaghetti code was a serious problem in the 1960's.  To improve readability, many programmers started to advocate complete avoidance of jump statements and introduced flags as the good-design alternative.

Nested Constructs

Enclosing one logic construct within another is called *nesting*.

Nested Selections

A selection within another selection is called a *nested selection*.

For example,

|  |
| --- |
| if (grade < 50) {  if (sup == 1)  printf("Sup\n");  else  printf("Failed\n");  } else  printf("Pass\n); |

Dangling Else

An ambiguity arises in a nested if else construct that contains an optional sequence (if).  Consider the following code snippet

|  |
| --- |
| if (grade < 50) {  if (sup == 1)  printf("Sup\n");  } else  printf("Pass\n); |

It is unclear to which if the else belongs if we remove the first pair of braces.

|  |  |
| --- | --- |
| // does else belong to first if?  if (grade < 50)  if (sup == 1)  printf("Sup\n");  else  printf("Pass\n); | // does else belong to second if?  if (grade < 50)  if (sup == 1)  printf("Sup\n");  else  printf("Pass\n); |

The C language always attaches the dangling else to the innermost if.

To associate an else with the next innermost selection, we retain the braces that wrap the innermost selection.

|  |
| --- |
| if (grade < 50) {  if (sup == 1)  printf("Sup\n");  } else {  printf("Pass\n);  } |

Nested Iterations

An iteration within another iteration is called a *nested iteration*.

The program below includes a nested iteration.  The output is shown on the right

|  |  |
| --- | --- |
| // Rows and Columns  // row\_columns.c  #include <stdio.h>  int main(void)  {  int i, j;  for (i = 0; i < 5; i++) {  for (j = 0; j < 5; j++) {  printf("%d,%d ", i, j);  }  printf("\n");  }  return 0;  } | 0,0 0,1 0,2 0,3 0,4  1,0 1,1 1,2 1,3 1,4  2,0 2,1 2,2 2,3 2,4  3,0 3,1 3,2 3,3 3,4  4,0 4,1 4,2 4,3 4,4 |