Every branching/looping pattern used in a piece of code adds a cognitive burden. Goto lets you create infinite such patterns, rapidly overwhelming people's ability to understand the code. Therefore, way back in the mists of time people (especially Dijkstra, 1968, Go to Statements Considered Harmful) settled on a standard set of control structures to replace most uses of goto, and that set has grown only slowly over time.

- Functions and procedures
- If/then/else and (sometimes) switch/case or case/in.
- While/for/do/repeat loops
- break, continue, and return statements
- Iterators, generators, list comprehensions (in some languages)
Spaghetti code is a pejorative phrase for source code that has a complex and tangled control structure, especially one using many GOTO statements, exceptions, threads, or other "unstructured" branching constructs. It is named such because program flow is conceptually like a bowl of spaghetti, i.e. twisted and tangled. Spaghetti code can be caused by several factors, such as continuous modifications by several people with different programming styles over a long life cycle.

Structured programming greatly decreases the incidence of spaghetti Code. When using the many forms of assembly language (and also the underlying machine code) the danger of writing spaghetti code is especially great.
Dijkstra argued that goto statements were harmful because they complicate two important and related tasks:

- Proving that a program fragment is correct
- Describing what a program has done so far.

As a simple example, consider a loop:

```c
while (n != 0)
{ /* do something */ }
```

Because this loop has `n != 0` as its condition, we know that when the loop terminates normally, `n` is zero. If there is a break statement inside the loop, it is possible that `n` will be nonzero when the loop terminates. This possibility shows how break statements can make it harder to understand what programs do.

How do we deal with the possibility of break statements in our programs when we analyse them? In one of two ways:

1. by including whatever conditions apply at the time the break is executed among the conditions that might apply when the loop terminates,
2. by ensuring that the loop-termination conditions are true when the break is executed.
Suppose we rewrite our while loop

```c
top: if (n == 0)
    goto bottom;
   /* do something */
    goto top;
bottom:
```

On the surface, this loop is no harder to analyze than the earlier version--after all, it behaves in the same way. However, when we added the label "bottom," we opened the possibility that a goto statement anywhere else in the program might jump there.

So now, instead of merely having to inspect the loop to understand what it does, we must inspect the entire program.
Sequence, Selection and Iteration

A sequence is one of the basic logic structures in computer programming. In a sequence structure, an action, or event, leads to the next ordered action in a predetermined order. The sequence can contain any number of actions, but no actions can be skipped in the sequence.

A selection (also called a decision) is also one of the basic logic structures in computer programming. In a selection structure, a question is asked, and depending on the answer, the program takes one of two courses of action, after which the program moves on to the next event. This structure is sometimes referred to as an if-then-else because it directs the program to perform in this way: If Condition A is True then perform Action X else perform Action Y.

An iteration is a single pass through a group/set of instructions. Most programs contain loops of instructions that are executed over and over again. The computer repeatedly executes the loop, iterating through the loop.