Effective application loops cases. Universal construction of a loop for programming languages.

Loop is one of the basic constructions of the existing programming languages and I will address myself on it in this article. Now there are a lot of languages that use various loops constructions. There are languages that use similar constructions and languages that use unique solutions. In the article you will look at some loops constructions, analyze their advantages and disadvantages, examine effective use cases of the loops and get acquainted with the universal loops construction. The article does not meant to be an exhaustive review of existing solutions but presents most of the common variants.

Loop is a programming language construction that allows executing the same piece of code several times. Such piece of a code is called the loop body. Execution of the loop body is called the iteration. Also there are few ways to specify the number of repetitions (iterations) of a loop: to provide the number of iterations explicitly; to specify some **exit condition** (loop is finished when the condition is met) or some **continuation condition** (loop is executed while the condition is met). Position of the condition can vary; for example, loops with pre- and post- conditions are frequently used. In these kinds of a loop a condition is checked before or after the iteration.

There are two approaches for a loop implementation. The first is having several loops constructions, each for concrete use case, so that the most suitable construction is used in each use case. The advantage is that the way to define the number of iterations and rules for building loop body can be different in various situations so it is very convenient to use a specific construction for a specific use case. Ambiguity is the disadvantage of this approach. Cases, when it is possible to use several constructions with the same result, arise quite often. If you look at smb. else code it is hard to guess why the particular construction has been used in this case. If you learn language it is also a bit difficult to keep in mind all possible loops. On the other hand there should be only one loop construction for all possible use cases. Unambiguity is the advantage in this case because the same task is always solved in the same way. The disadvantage is that in some cases we should write a lot of complex code to make it possible to use the construction.

Let us look at both approaches. I will use C++(very similar loops exist in c# and java) for the first approach and Eiffel for the second.

C++[1] has three kinds of loops:

1. **while** (<condition>) { <body> }

   This loop has continuation condition. Condition is checked before every iteration. The braces can be skipped in case loop body consists of one operator.

2. **do** { <body> } **while** (<condition>);

This loop also has continuation condition. But, opposite to **while** loop, the condition is checked after each iteration. The braces can be skipped in case loop body consists of one operator.

3. **for** (<initialization>; <condition>; <incrementation>) { < body > }

   This loop has continuation condition that is checked before every iteration. Also there are initialization and iteration sections. The initialization section contains code that is executed once before very first iteration, Iteration section contains code that is executed after each iteration and before checking continuation condition. The braces can be skipped in case loop body consists of one operator.

   All described constructions can also have **break** and **continue** operators. **break** allows stopping loop execution. **continue** allows finishing current iteration and starting next one.

   In appendix A, as well, there are several exotic ways of loop creation in C++.

The Eiffel [2] has only one loop construction:

1. **from** <initialization> **until** <condition> **loop** < body> **end**

   Before describing this construction I would like to mention two optional sections - **invariant**, **variant** in Eiffel loops. These constructions are for checking loop correctness. In the article I describe some possible variants of loops sections organization and consecution of their execution. So I do not examine these sections (corresponding information can be found at B. Meyer book [3])

   This loop has initialization section and exit condition. Code from initialization section is executed once before first iteration. Exit condition is checked before every iteration.

Let us look now at the common task – reading from file, symbol by symbol. For simplicity I assume that file is already open for reading (`FILE *file` in C++ and `file:KL_TEXT_INPUT_FILE` in Eiffel)

1. **while** from C++

```
int symb = getc (file);

while(symb != EOF)
    {
    // do anything

    symb = getc(file);
    }
```

2. **do while** from C++

```
int symb;

do
    {
    symb = getc(file);

    if(symb == EOF) { break; }

    // do anything
    }
while(symb != EOF);
```

3. **for** from C++

```
for (int symb = getc(file);
     symb != EOF;
     symb = getc(file)
    )
    {
    // do anything
    }
```

4. **loop** from Eiffel

```
from
    file.read_character
until
    file.end_of_file
loop
    -- do anything

    file.read_character
end
```

All loops do the same job: read a character from file, check for "an end of a file" condition, in case an end of file is achieved a loop is finished, otherwise execute some code (do anything) and start new iteration.

No one can do the job well because all loops have code duplication. The code duplication is marked with color. In the examples the duplication is minimal, but even in this case it can be a source of problems (e.g. if a developer modifies code in one place and forgets to do the same in another). In the real life there can be more complicated cases if the amount of duplication is much larger. Note that different places where condition is checked in loops **while** and **do while** lead to different code duplication - code for reading symbol from the file is duplicated in **while** loop whereas in **do while** loop the exit condition is duplicated. Avoiding this duplication is possible only by changing loop structure. The same method can be used for all C++ loops.

1. **while from C++**

```
int symb;
while (true)
    {
    symb = getc(file);

    if(symb == EOF) {break;}

    // do anything
    }
```

2. **for from C++**

```
int symb;
for (;true;)
    {
    symb = getc(file);

    if(symb == EOF) {break;}

    // do anything
    }
```

I avoid usual check of continuation condition and added new check of exit condition using **break** operator. It is not possible to do the same in Eiffel loops because of lack of the operator that can stop the loop. So in Eiffel it is not possible to avoid duplication. You can only rewrite it so that the check of exit condition will be duplicated instead of reading a symbol.

```
is_end_of_file : BOOLEAN
from
    is_end_of_file := False
until
    is_end_of_file
loop
    file.read_character
    is_end_of_file := file.end_of_file

    if not is_end_of_file then
        -- do anything
    end
end
```

So, as it is shown, there are situations when it is necessary to check if there is an end of a loop not only before or after loop body but also inside the body. All examined constructions do not support this directly but C++ allows doing this using additional operators.

Now let us write a loop that reads html tag from a file. I assume that the file is already opened for reading, html tag is the sequence of symbols started from '<' and finished with '>', symbol '>' can not be a part of a tag, the current position in the file is set at the beginning of a tag. The task is to read all symbols from the current position till '>' symbol inclusively. In case you meet the end of the file before a tag was read it is necessary to display a error message.

1. **while** from C++

```
int symb = getc (file);
std::string tag;
bool tag_was_read = false;

while(symb != EOF && !tag_was_read)
    {
    tag += symb;

    if (symb != '>')
        { symb = getc(file); }
    else
        { tag_was_read = true; }
    }

if ( symb == EOF )
    { printf("Error! Tag is invalid!"); }
```

2. **do while** from C++

```
int symb;
std::string tag;

do
    {
    symb = getc(file);

    if(symb != EOF)
        { tag += symb; }
    }
while(symb != '>' && symb != EOF);

if ( symb == EOF )
    { printf("Error! Tag is invalid!"); }
```

3. **for** from C++

```
int symb;
std::string tag;
bool tag_was_read = false;

for (symb = getc(file);
     symb != EOF && !tag_was_read;
    )
    {
    tag += symb;

    if (symb != '>')
        { symb = getc(file); }
    else
        { tag_was_read = true; }
    }

if ( symb == EOF )
    { printf("Error! Tag is invalid!"); }
```

4. **loop** from Eiffel

```
tag_was_read : BOOLEAN
tag : STRING

from
    tag_was_read := false
    file.read_character
until
    file.end_of_file or else tag_was_read
loop
    tag.extend ( file.last_character )

    if not file.last_character.is_equal('>')
    then
        file.read_character
    else
        tag_was_read := true
    end
end

if file.end_of_file then
   io.put_string("Error! Tag is invalid!")
end
```

All loops do the same sequence of actions:

1. Read a symbol from a file
2. Check for the end of a file. If the end of a file is met then stop the loop and go to step 5, otherwise go to the next step.
3. Copy the read symbol to a buffer.
4. Check for the end of a tag. In case of the end of a tag stop the loop and go to step 5, otherwise start a new iteration from step 1.
5. Check for the end of a file. In case of the end of a file display an error message.

The specific feature of this algorithm is the necessity to have several points to stop the loop and to link a processing with one of these points. Since the loop cannot be stopped as soon as a tag is read I have to add an extra conditional operator to suppress a part of the loop body. And since I cannot link an error message display with the end of a file condition I have to add an extra conditional operator after the loop to check which condition has stopped the loop. As a result all loops contain duplicated code (marked with color).

But it is still possible to implement loops in C++ effectively. For this you should use operator **break** to stop the loop and to move an error message display inside the loop body exactly before the exit of the loop.

1. **while** from C++

```
int symb;
std::string tag;

while(true)
    {
    symb = getc (file);

    if (symb == EOF)
        {
        printf("Error! Tag is invalid!");
        break;
        }

    tag += symb;

    if (symb == '>')
        {break;}
    }
```

2. **do while** from C++

```
int symb;
std::string tag;

do
    {
    symb = getc (file);

    if (symb == EOF)
        {
        printf("Error! Tag is invalid!");
        break;
        }

    tag += symb;
    }
while (symb != '>');
```

Now all loops do not contain duplicated code. To perform the task properly is possible only by changing rules of loop construction. Operator **break** that allows stopping the loop is the most useful. In Eiffel I cannot avoid code duplication because of lack of operators to stop the loop.

At the same time there are well known disadvantages of using operators **break** and **continue**. First of all it is their ambiguity. It is possible to ignore standard rules of a loop construction and to implement the same rules using these operators. This duality confuses a developer because he should analyze why this is done in that way. Second, these operators can be nested in other operators so that it is hard to understand which condition breaks the loop and what actions are executed during iteration. So it would be nice to find the way to perform the task without these disadvantages. Note, that if it is allowed to use **continue** only in one operator (level of nesting would be 1) then the operator **continue** will be analogous to a conditional operator with inverted condition.

```
int symb = getc (file);
```

```
while(symb != EOF)
```

```
{
symb = getc (file);

if (symb == ' ')
    { continue; }

// do something
}
```

```
int symb = getc (file);
while(symb != EOF)
    {
    symb = getc (file);

    if (symb != ' ')
        {
        // do something
        }
    }
```

Let us examine the possibilities that are missed in the analyzed examples:

1. The possibility to have an exit point in any place inside the loop body.
2. The possibility to have several exit points.
3. The possibility to link a processing with any exit point.
4. Clear understanding of exit conditions and of sequence of actions in the loop body

Perhaps, if these possibilities are added in a standard construction it could be possible to use one universal loop construction for all use cases. Fig 1.demonstrates the block diagram of such construction:

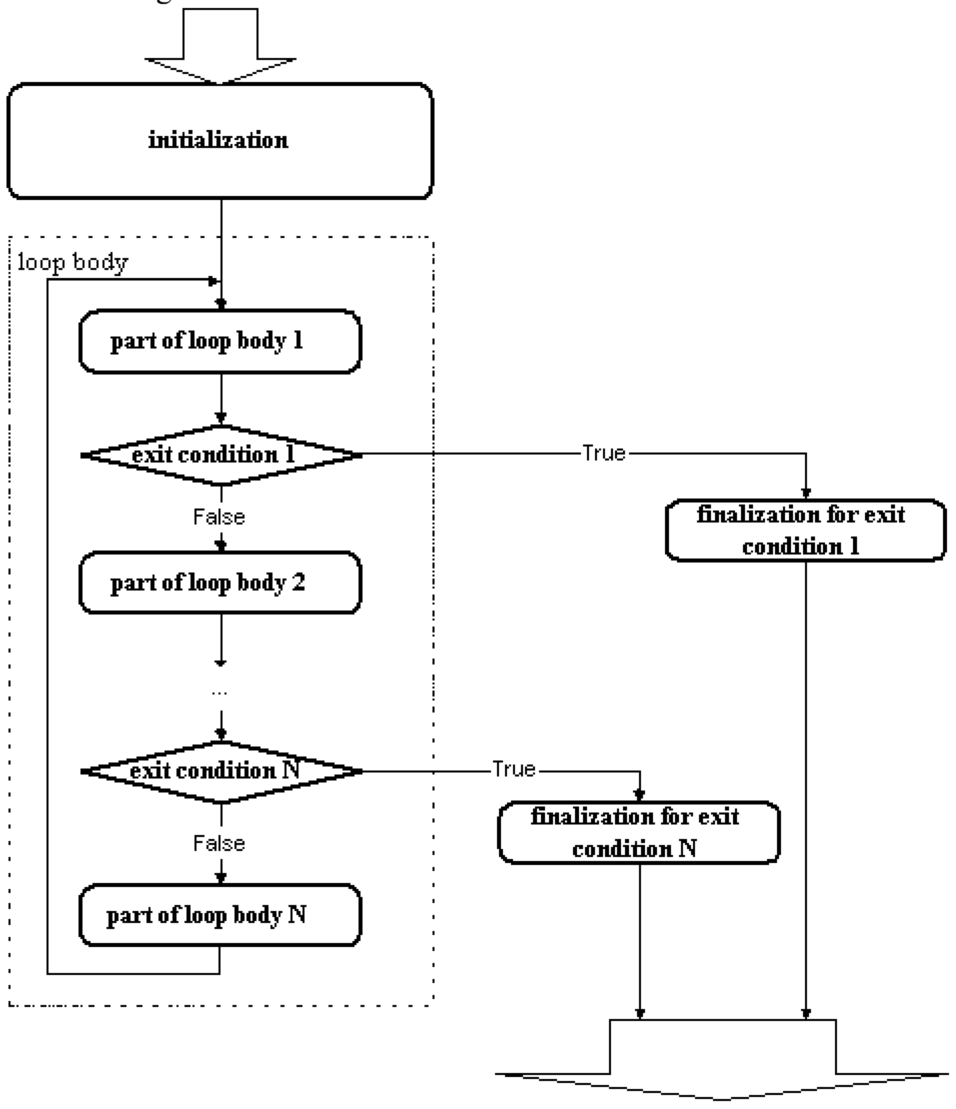

Fig.1

The construction defining an exit point cannot be nested inside other constructions (unlike **break**). Conditional operator will be used to miss a part of the loop body in a particular iteration (instead of **continue**). According to the described model the loop construction is the following:

C++ like

```
loop ( /* initialization*/ )
  {
   /* loop body part 1 */
  when (  /* exit condition */ )
     do { /* handler */ } exit;
   …..
   /* loop body part N */
  }
```

loop with braces defines loop body.
when do exit specify exit condition, **do** with braces is optional.
do specifies a handler for particular exit point.

Eiffel like

```
from
   -- initialization
loop
   -- loop body part 1
   until  -- exit condition
      on_exit
          -- handler
      end
   ……
   -- loop body part N
end
```

loop .. **end** defines loop body.
until on_exit end specify exit condition, **on_exit** is optional.
on_exit specifies a handler for particular exit point.

The loop has several exit conditions and they can be placed in any point of the loop body. It is possible to link a handler to any exit point.

In possible C++ compiler the examined earlier example could look like as the following:

```
std::string tag;

loop ( int symb; )
    {
    symb = getc (file);

    when (symb == EOF) do { printf("Error! Tag is invalid!"); } exit;

    tag += symb;

    when (symb == '>') exit;
    }
```

Note that keyword **exit** is not the operator in contrast to **break**, so **exit** cannot be nested inside other operators.

In patched Eiffel compiler the examined earlier example look like this:

```
tag : STRING

from
loop
    file.read_character
until
    file.end_of_file on_exit io.put_string("Error! Tag is invalid!") end

    tag.extend ( file.last_character )
until
    file.last_character = '>' end
end
```

Apparently, this construction allows avoiding code duplication and ambiguity. This loop construction was implemented for open source Eiffel compiler – SmartEiffel [4] and can be downloaded from the sourceforge site [8].

All described problems are not critical in the sense that it is always possible to solve any of them using more complex or less clear construction. But even these small problems can create noticeable complexity for software development, taking time to write duplicated code, increasing risk of errors and decreasing code readability.

Most of the described above ideas you can meet in non-mainstream programming languages. So Ada has **dowhiledo** loop which allows specifying an exit condition in any place of the loop body. The same possibility exists in such languages as L76 [5], Q [6]. Similar problems are also regularly discussed in various news groups. I consider the article summarizes information on the topic.

Literature

[1] B. Stroustrup. The C++ Programming Language (Special 3rd Edition) Addison-Wesley, 2000.
[2] B. Meyer. Eiffel : The Language, Prentice Hall, 1991.
[3] B. Meyer. Object-Oriented Software Construction(2nd Edition), Prentice Hall, 2000.
[4] http://smarteiffel.loria.fr/
[5] V. Pentkovsky. Programming Language L76, Nauka, 1989.
[6] http://www.q-software-solutions.com/q/
[7] E. Dijkstra. Go To Statement Considered Harmful. Journal *Communications of the ACM*, Vol. 11, No. 3, March 1968, pp. 147-148
[8] http://uni-loop.sourceforge.net

Appendix A

1. Loop **while** from the first example could be written as following:

```
int symb;
while((symb=getc(file))!= EOF)
    {
    // do anything
    }
```

```
int symb;
while(symb=getc(file), symb != EOF)
    {
    // do anything
    }
```

This allows avoiding code duplication. But this method can be used only in simple cases because moving a part of the loop body into condition makes it harder to understand the loop. Moreover, in complex cases it will be just impossible to move all complicated code to loop condition.

2. It is possible to use **goto** instead of **break**.

```
int symb;
do
    {
    symb = getc(file);

    if(symb == EOF) { goto exit_loop; }

    // do anything
    }
while(symb != EOF);
```

```
exit_loop:
```

Well-structured programs should avoid using **goto** [7].

3. Using exceptions for exit from a loop:

```
int symb;
try
    {
    do
        {
        symb = getc(file);

        if(symb == EOF) { throw 0; }

        // do anything
        }
    while(symb != EOF);
    }
catch ( int )
    {  }
```

I think that using exceptions to exit from a loop can be considered only as an exotic method, since this manner do not add any advantages compared to others but requires more coding.