COSC 30403

Programming Language Concepts

Chapter 6 - Control Flow

Exercises: 6.4, 6.8, 6.13, 6.16, and 6.25

Submitted by: Rahul Shrestha
Submitted to: Dr. Mei

Date: April 2, 2025

6.4 Translate the following expression into postfix and prefix notation:

$$\frac{-b + \sqrt{b \times b - 4 \times a \times c}}{2 \times a}$$

Do you need a special symbol for unary negation?

Answer:

Postfix: b neg b b * 4 a * c * - sqrt + 2 a * /
Prefix: / + neg b sqrt - * b b * 4 * a c * 2 a

In postfix notation, it's acceptable to use the standard `-` symbol instead of `neg` because it appears immediately after `b`, making it clearly a negation rather than a subtraction. The same applies to prefix notation—using `-` won't cause confusion. Therefore, while `-` is valid in both cases, using `neg` can improve clarity.

6.8 Languages that employ a reference model of variables also tend to employ automatic garbage collection. Is this more than a coincidence? Explain.

Answer:
Yes, it is more than a coincidence. Languages that implement a reference model—where variables point to memory locations rather than store data directly—often adopt automatic garbage collection. This design simplifies managing complex data structures and ensures that once an object is no longer referenced, the garbage collector can reclaim its memory. By combining references with automated collection, these languages reduce memory leaks, minimize manual memory management errors, and make programming more reliable.

6.13 Neither Algol 60 nor Algol 68 employs short-circuit evaluation for Boolean expressions. In both languages, however, an `if...then...else` construct can be used as an expression. Show how to use `if...then...else` to achieve the effect of short-circuit evaluation.

Answer:
The syntax of `if...then...else` is
*If **condition** then*
[statements]
else
[statements]

In nested loops, the `else` is always associated with the nearest preceding `if`.
The nested loops are italicized in the below code.

A and B using if...then...else

If A then *// when A is true*
If B then *// if B is true*
return True; *// return true*
else
return False; *// if B is not true, return false*

else
return False; // if A is not true, return false

A or B using if...then...else

If A then *// if A is true*
return True; // return true

else *// if A is not true*
If B then *// if B is true*
return True; // return true
else
return False; // if B is not true, return false

6.16 The equivalence of `for` and `while` loops, mentioned in Example 6.64, is not precise. Give an example in which it breaks down. Hint: think about the `continue` statement.

Answer:

The equivalence between `for` and `while` loops in C breaks down when using the `continue` statement.

Why? In a `for` loop, the increment expression (`i += step`) is executed automatically after the loop body, even when `continue` is used. But in a `while` loop, the increment must be explicitly placed. If it's skipped due to `continue`, it can lead to unexpected behavior, like an infinite loop.

Example using a `for` loop:

```
for (int i = 0; i < 5; i++) {
    if (i == 2)
        continue;
    printf("%d ", i);
}
```

Output:

0 1 3 4

Even when `continue` is triggered, the loop increments `i` automatically.

Now consider the equivalent `while` loop:

```
int i = 0;
while (i < 5) {
    if (i == 2)
        continue; // Infinite loop!
    printf("%d ", i);
    i++;
}
```

Issue: When `i == 2`, the `continue` skips the increment, causing `i` to stay at 2 forever. This leads to an infinite loop.

Corrected `while` version:

```
int i = 0;
while (i < 5) {
    if (i == 2) {
        i++; // Fix: increment before continue
        continue;
    }
    printf("%d ", i);
    i++;
}
```

Output:

0 1 3 4

Conclusion: The presence of `continue` breaks the naive equivalence between `for` and `while` loops because the increment in a `for` loop is guaranteed, while in a `while` loop, it must be explicitly written.

6.25 Consider a mid-test loop, here written in C, that looks for blank lines in its input:

```
for (;;) {
    line = read_line();
    if (all_blanks(line)) break;
    consume_line(line);
}
```

Show how you might accomplish the same task using a `while` or `do` (repeat) loop, if mid-test loops were not available. (Hint: One alternative duplicates part of the code; another introduces a Boolean flag variable.) How do these alternatives compare to the mid-test version?

Explanation:

The original loop reads a line, checks if it's all blanks, breaks if true, else consumes the line. It's a loop with a mid-test condition, as the decision to break out of the loop or continue is determined in the middle of the loop's body.

We can rewrite this with a `while` loop or a `do-while` loop in two main ways:

- By duplicating some part of the code.
- By introducing a Boolean flag.

Using while Loop with Code Duplication:

```
char* line;

// Read the first line outside the loop to initiate the process.
line = read_line();

// Continue looping until the line is all blanks.
while (!all_blanks(line)) {
    consume_line(line);
    line = read_line();
}
```

Using while Loop with Boolean Flag:

```
char* line;
bool keepReading = true;

while (keepReading) {
    line = read_line();

    // If the line is all blanks, set the flag to false to break the loop
    if (all_blanks(line)) {
        keepReading = false;
    } else {
        consume_line(line);
    }
}
```

Using do-while Loop with Code Duplication:

```
char* line;

do {
    line = read_line();
    if (all_blanks(line)) break;
    consume_line(line);
} while (true);
```

Answer

Output:

Comparison:

Code Duplication: Using code duplication can lead to harder maintenance. If, for example, the method of reading a line changes, you have to update it in multiple places. It's generally recommended to avoid code duplication.

Boolean Flag: Introducing a flag can make the logic of the loop clearer. But, it also introduces an extra variable that needs to be managed.

The mid-test loop version is more concise and clear. The decision to continue or break from the loop is made in a single place within the loop body.

It's a cleaner solution than either of the alternatives. However, each method has its use-cases, and the best choice depends on the specifics of the problem and personal preference.