CPTS 121 L25

LAB 6: ITERATIVE STATEMENTS

FOR LOOPS

- Three parts to a for loop: initializer, condition, incremental statements
- GENERALLY: Use when we know how many times to do something, or if we need to progress in a list.
- Variable can be declared in the init value, so it is only visible within the loop.
- Use a comma between multiple init or increment statements.

```
for (initialization; condition; increment) {
    statements
}

for (int i = 0; i < 100; i += 1) {
    // do something
}
```

HOW FOR LOOPS WORK

- Assume we have the below for statement. When the computer compiles the for loop, a similar code is constructed in assembly.
  - For simplicity, I wrote it in C
  - IMPORTANT: NEVER USE GOTO STATEMENTS!!!

```
17      for (int i = 0; i < 100; i += 1) {
18          sum += 1;
19      }
```

```
16      int i;
17      int sum = 0;
18
19  INITALIZERS:
20      i = 0;
21      goto CONDITION;
22  BODY:
23      sum += i;
24      goto INCREMENT;
25
26  INCREMENT:
27      i += 1;
28      goto CONDITION;
29
30  CONDITION:
31      if (i < 100) {
32          goto BODY;
33      } else {
34          goto EXIT_LOOP;
35      }
36
37  EXIT_LOOP:
38      printf("Sum is: %d", sum);
39      return 0;
40  }
```

HOW FOR LOOPS WORK (ADVANCED)

Purple arrow means branch.
C equivalent is goto.

Orange arrow is checking if w8 is >= 100. It stores result in w8

Blue arrow is checking if the condition was true. If so, jump to LBB0_4

Note: w8 is a register; think of this as a variable for now

- Assume we have the below for statement. When the computer compiles the for loop, this is what an M1 Max Mac will compile.

```
17      for (int i = 0; i < 100; i += 1) {
18          sum += 1;
19      }
```

```
18      b   LBB0_1
19  LBB0_1:
20      ldur    w8, [x29, #-12]
21      subs    w8, w8, #100
22      cset    w8, ge
23      tbnz    w8, #0, LBB0_4
24      b   LBB0_2
25  LBB0_2:
26      ldur    w8, [x29, #-8]
27      add w8, w8, #1
28      stur    w8, [x29, #-8]
29      b   LBB0_3
30  LBB0_3:
31      ldur    w8, [x29, #-12]
32      add w8, w8, #1
33      stur    w8, [x29, #-12]
34      b   LBB0_1
35  LBB0_4:
36      ldur    w9, [x29, #-8]
37
```

WHILE LOOPS

```
18      while (i < 100) {
19          i += 1;
20      }
```

- We want to use this when we do not necessary know when it will end.
- It must be a finite condition, but we are less likely to know how many iterations there will be.
- **Use this when we want to check the condition before running.**

```
18  CONDITION:
19      if (i < 100) {
20          goto BODY;
21      } else {
22          goto EXIT_LOOP;
23      }
24
25  BODY:
26      i += 1;
27      goto CONDITION;
28
29  EXIT_LOOP:
30      printf("i: %d", i);
31      return 0;
```

WHILE LOOPS (ADVANCED)

C

```c
    while (i < 100) {
        i += 1;
    }
```

Assembly

```
LBB0_1:
    ldr w8, [sp, #8]
    subs    w8, w8, #100
    cset    w8, ge
    tbnz    w8, #0, LBB0_3
    b   LBB0_2
LBB0_2:
    ldr w8, [sp, #8]
    add w8, w8, #1
    str w8, [sp, #8]
    b   LBB0_1
LBB0_3:
    ldr w9, [sp, #8]
```

Orange arrow is checking if w8 is >= 100. Stores result in w8

Add 1 to w8

Purple arrow means branch. C equivalent is goto.

C, using goto

```c
CONDITION:
    if (i < 100) {
        goto BODY;
    } else {
        goto EXIT_LOOP;
    }

BODY:
    i += 1;
    goto CONDITION;

EXIT_LOOP:
    printf("i: %d", i);
    return 0;
```

DO-WHILE LOOPS

- We want to use this when we do not necessary know when it will end.
- It must be a finite condition, but we are less likely to know how many iterations there will be.
- **Use this when we DO NOT want to check the condition before running.**

```
18      do {
19          i += 1;
20      } while (i < 100);
```

```
22  BODY:
23      i += 1;
24      goto CONDITION;
25
26  CONDITION:
27      if (i < 100) {
28          goto BODY;
29      } else {
30          goto EXIT_LOOP;
31      }
32
33  EXIT_LOOP:
34      printf("i: %d", i);
35      return 0;
```

DO-WHILE VS WHILE LOOPS

```
18      while (i < 100) {
19          i += 1;
20      }
```

- Use a do-while when we need **to run something before** checking a condition, every time. Will run 0+ times
- Use a while loop when we need to **check the condition before** running something. Will run 1+ times.

```
18      do {
19          i += 1;
20      } while (i < 100);
```

CHANGE FROM A FOR LOOP TO A WHILE LOOP [+1 PT]

```
6       for (int i = 0, p = 0; // Initialization
7            i < 100; // condition
8            i += 1, p -= 1) { // increment
9           p += i; // statements
10      }
```

```
6       for (int i = 0, p = 0; i < 100; i += 1, p -= 1) {
7           p += i;
8       }
```

CHANGE FROM A FOR LOOP TO A WHILE LOOP (SOLN) [+1 PT]

```
9       int i = 0, p = 0;
10
11      while (i < 100) {
12          p += i;
13
14          i += 1;
15          p -= 1;
16      }
```

CHANGE FROM A FOR LOOP TO A WHILE LOOP (ADVANCED SOLN HINT)

(i++) will return i's previous value plus one; i++ is invalid for this context.

(x = <SOME MATH>) will assign x to the value returned by the formula.

y is logically true such that y IS NON-ZERO

Assume:

- i = 1 at start
- y = 1
- i can count to infinity

while ((i++) && y) { … }

This statement will run forever.

RUN 0:
    while (1 && 1)

RUN 1:
    while (2 && 1)

RUN 2:
    while (3 && 1)

RUN 3:
    while (4 && 1)

RUN 4:
    while(5 && 1)

RUN N:
    while(N && 1)

CHANGE FROM A FOR LOOP TO A WHILE LOOP (ADVANCED, NON-OPTIMAL, SOLN) [+2 PTS]

```
12      while (i < 100 && ((p += i) || 1) && ((p--) || 1) && (++i));
```

i < 100: Obvious; this is the condition we want.

&& (p += i || 1): p += i must be in (), that way it will give us the result of p += i. That is, it will give the new value of p. We need to add || 1 since p may become 0, but we still need to continue.

&& ((p--) || 1): p-- must be in (), so that the new value is used. We || 1 since p may be zero, but we still need to run.

&& (++i): We MUST use ++i since the value is initially 0, which means the loop will run 0 times. By using ++i, it will increment i then use the value (this will increment so that we have i = 0..<100.)

This is a very poor solution for deployment. I am only showing this because it shows understanding of operators.

CHANGE FROM A WHILE LOOP TO A FOR LOOP [+1 PT]

```
5       int x = -10, y = 200;
6
7       while (x < 10 && y > 100) {
8           x += 1;
9           y -= 10;
10      }
```

CHANGE FROM A WHILE LOOP TO A FOR LOOP (SOLN) [+1 PT]

```
13      for (int x = -10, y = 200;
14           x < 10 && y > 100;
15           x += 1, y -= 10);
```

```
13      for (int x = -10, y = 200; x < 10 && y > 100; x += 1, y -= 10);
```