CSE 2421

The C Language – Part 3

Overview

- Loop constructs in C
  - for loops
  - while loops
  - do while loops
  - break and continue
- Conditional statements in C
  - if
  - switch-case
- Boolean issues
- The comma operator
- Enumerated data types

Loop Constructs

```
for (expression1; expression2; expression3)
     { statement(s); }
while (expression)
     {statement(s); }
do {
      statement(s);
} while (expression);
```

- Statement(s) only execute when expression (expression2 in **for loop**) is non-zero (TRUE).
- Notice the semi-colon locations.
- do while {statement(s)} **always** execute **at least once** (not necessarily true for the other two constructs).

for loop

```
for (expression1; expression2; expression3)
      { statement(s); }
```

- expression1 is *the initialization*, and is evaluated only once before expression2 is evaluated the first time.
- expression2 is *the condition*, and is evaluated once before each execution of the body ({statement(s)}); if it evaluates to a non-zero value, the body is executed; otherwise, execution of the for-loop terminates, and execution continues with the statement following the for-loop.
- expression3 is called *the adjustment*, and is evaluated after all statements in the body are executed, and just before the condition is evaluated again.

for loop example

```
#include <stdio.h>

int main ()
{
     int n, sum = 0;    /*defines n and sum; initializes sum*/
     /* n=n+1 could be n++ or n+=1 or ++n */
     for (n = 1; n <= 10; n = n + 1)
     {
           sum = sum + n;
     }
     printf("The sum of integers from 1 to 10 is %d\n", sum);
     return 0;
}
```

while loop

```
while (expression)
    {statement(s) }
```

- expression is evaluated; if it evaluates to non-zero (TRUE), {statement(s)} is/are executed, and expression is evaluated again…
- If expression evaluates to zero (FALSE), execution of the while loop terminates, and the statement following the loop is executed (the next statement after the }).

while loop example

```
#include <stdio.h>
int main ()  {
    int n = 1, sum = 0;
    while (n <= 10)
    {
        sum = sum + n;
        n = n + 1;
    }
    printf("\n The sum of integers from 1 to 10 is %d\n", sum);
    return 0;
}
```

while loop example with an embedded assignment expression

```
#include <stdio.h>
int main()
{     char ch;
      printf(“Please enter characters one at a time, followed by enter.
When you want to stop, enter the # character followed by enter:\n”);
      while ((ch = (char)getchar()) != ‘#’)
            {
            putchar(ch);
            }
        return 0;
}

/*What does this simple program do?*/
/*What doesn’t it do?  EOF???      */
```

do while

```
do {
        statement(s);
} while (expression);
```

- statement(s) are executed once, and then expression is evaluated; if it evaluates to non-zero (TRUE) {statement(s)} are executed again, until expression evaluates to zero (FALSE).
- The result is that {statement(s)} will always be executed at least once, even if expression is false before the first execution.

do while loop example

```
#include <stdio.h>
int main()
{       int print_it = 0;
         do {
              printf(“Hello World!\n”);
         } while (print_it != 0);
         return 0;
}
/* Hello World! is printed once, even though print_it
is zero before the loop body is executed. */
```

1 line loop bodies

- If a loop statement is only going to contain one line you can leave off the enclosing {}

```
while(x < 10) printf("%d", x++);
```

```
for(int i = 0; i < 10; ++i)
    printf("%d", i);
```

Nonetheless, best practice is to use {} for clarity (or in case you need to add additional statements to the body of the loop later), but this shorthand is acceptable. Consider single line.

Break, Continue

- Keywords that can be very important to looping are *break* and *continue*.
- *break* exits the innermost loop or switch statement (see below) which encloses it regardless of loop conditions
- *continue* causes the loop to stop its current iteration, do the adjustment (in the case of FOR loops) and begin to execute again from the top. It may only occur inside of for, while, or do while loops.

Jump Statement

- **goto** plus a labeled statement
  - **goto** identifier; /* identifier is called *a label* here */
  - identifier: statement;
- Don't have to pre-define or declare the identifier
- Identifiers **must be unique**
- A statement label is meaningful only to a **goto** statement, otherwise it's ignored
- Both the keyword **goto**, and the label, must appear in a single function (i.e., you can't jump to a statement in a different function)
- Identifier labels have their own name space so the names do not interfere with other identifiers (however, for clarity, identifiers that are the same as variable or function names should not be used!).
- Best practice to use break, continue, and return statement in preference to **goto** whenever possible (i.e., virtually always).
  - Special case where **goto** is considered acceptable: exiting multiple levels of loops (a deeply nested loop).

Break Example

```
/* playing checkers */

while (1)
{
      take_turn(player1);
      if((has_won(player1) || (wants_to_quit(player1) == TRUE)){
                  break;
      }
      take_turn(player2);
      if((has_won(player2) || (wants_to_quit(player2) == TRUE)){
            break;
      }


}
```

Continue Example

```
/* playing monopoly */

for (player = 1; has_won() == 0; player++)
{
     if (player > total_number_of_players) {
              player = 1;
     }
     if (is_bankrupt(player)) {
              continue;
     }
     take_turn(player);
}
```

goto Example

```
#include <stdio.h>
int main() {
     int i, j;
     for ( i = 0; i < 10; i++ )
     {
               printf("Outer loop. i = %d\n",i);
               for ( j = 0; j < 3; j++ )
               {
                       printf("Inner loop. j = %d\n",j);
                       if ( i == 1 )
                               goto stop;
               }
       }
     /* This message does not print: */
     printf("Loop exited. i = %d\n", i );
  stop:
       printf( "Jumped to stop. i = %d\n", i );
     return 0;  }
Output:
?????
```

Goto Example

```
#include <stdio.h>
int main() {
     int i, j;
     for ( i = 0; i < 10; i++ )
     {
                    printf("Outer loop. i =%d\n",i);
                    for ( j = 0; j < 3; j++ )
                    {
                               printf("Inner loop. j = %d\n",j);
                               if ( i == 1 )
                                          goto stop;
                    }
          }
     /* This message does not print: */
     printf("Loop exited. i = %d\n", i );
  stop:
         printf( "Jumped to stop. i = %d\n", i );
     return 0;  }
Output:
Outer loop. i = 0
Inner loop. j = 0
Inner loop. j = 1
Inner loop, j = 2
Outer loop. i = 1
Inner loop, j = 0
Jumped to stop. i = 1
```

If, else-if, switch-case conditional statements

```
if ( expression ) {
     /* Execute these  stmts if
      expression != 0 */ }
else {
     /* Execute these stmts if
 expression == 0 */ }
```

```
if (expression1) {
      statement(s);  }
else if (expression2) {
      statement(s);  }
else {
      statement(s);  }
```

```
switch ( <variable> ) {
     case value1:              /* Note - use : not  ; */
          Code to execute if <variable> == value1;
          break;
     case value2:
          Code to execute if <variable> == value2;
          break;
...  default:
          Code to execute if <variable> does not equal
           the value following any of the cases… break; }
```

- SWITCH NOTES:
- Notice, no {} blocks within each case!
- Notice the colon for each case and value.
- value1, value2, etc. in a switch statement must be constant values; variable should be some integer type
- The default case is optional, but it is wise to include it as it handles any unexpected cases.
- Fall-throughs are allowed (if break is omitted)
- Chooses first value that matches…

Dangling else

```
/* Suppose this code, with scores from 0 to 100 */

int score = 80;

if (score >= 60)
        if (score >= 90)
                printf(“Congratulations, you got an A!\n”);
else printf(“Unfortunately, that’s not passing!\n”);



/* Output? */
```

Dangling else – output

Output:

```
Unfortunately, that's not passing!
```

What happened? We only wanted this message for scores < 60.

Dangling else – output

- The problem is that the compiler does not pay attention to formatting, so even though the **else** is aligned with the first **if**, the compiler pairs it with the second **if**.
- Rule: The compiler pairs any **else** with the closest (most recent) unmatched **if** which is not enclosed in a different block.
- "Unmatched" here means that the **if** has not already been paired with an **else**.

Dangling else – how to fix it

```
int score = 80;

if (score >= 60)
{
        if (score >= 90)
                printf(“Congratulations, you got an A!\n”);
}
else printf(“Unfortunately, that’s not passing!\n”);

/* Now, the else will be paired with the first if, because the second one
is enclosed in a different block. */
```

else-if Example

```
#include <stdio.h>
int main()    {
      int age;
      printf( "Please enter your age" );
     scanf( "%d", &age );
      if ( age < 100 ) {
             printf ("You are young!\n" );         }
      else if ( age == 100 ) {
             printf("You are old\n" );
      }
      else  {
             printf("You are really old\n" );      }
     return 0;
}
```

Switch Case

```
switch ( x ) {
case 'a':
       /* Do stuff when x == 'a' */
       break;
case 'b':
case 'c':
case 'd':
       /* Fall-through technique...
            cases b,c,d all use this code          */
       break;
default:
       /* Handle cases when x is not
            a,b,c or d. ALWAYS have a
            default  case*/
       break; /* this break is not necessary, but legal */
}
```

Boolean Issues

- Every boolean test is an implicit comparison against zero (0).
- However, zero is not a simple concept. It represents:
  - the integer 0 for all integer types
  - the floating point 0.0 (positive or negative)
  - the null character (‘\0’)
  - the null pointer
- In order to make your intentions clear, explicitly show the comparison with zero for all scalars, floating-point numbers, and characters.

Boolean Issues

- **int i; if (i)** is better represented as **if (i != 0)**
- **float x; if (!x)** is better represented as **if (x == 0.0f)**
- **char c; if (c)** is better represented as **if (c != '\0')**

- An exception is made for pointers, since 0 is the only language-level representation for the null pointer.

- The symbol NULL is not part of the core language – you have to include a special header file to get it defined (stdlib.h or stddef.h). More on this later.

Boolean issues

- To write an INFINITE LOOP
  - for (;;) ...
  - while (1) ...
- The former is more visually distinctive, but both forms are used.
- In industry code that I saw, while(1) was used almost exclusively

The Comma Operator

- Used to link related expressions together
- Evaluated from left to right
- The value of the right-most expression is the value of the whole combined expression
- It forces a sequence point – all side effects are settled
- Parentheses are necessary; if no parentheses surround an expression with commas, the commas are *separators*, and not the comma operator!
- Example:
  - i = a, b, c;   /*stores a into i – commas here are separators*/
  - i = (a, b, c);   /*stores c into i – these commas are comma operators*/
- For loop:
  - for (n=1, m=10; n<=m; n++, m--)
- While:
  - while (c=getchar(), c!= ‘0’)
- Exchanging values:
  - (t=x, x=y, y=t); /*Better not to use comma operator unless you want to use this as a boolean, for example, to control a loop or an if or if-else conditional*/

Student Bug of the Semester, SP 2020

```
#include <stdio.h>
int main()
{
        /* we leave controlled airspace when we go above 50,000 feet */
        double alt = 65750.0;

        /* what prints? */
        if(alt > 50,000)
        {
            printf("Above 50k\n");
        }
        else
        {
            printf("At or below 50k\n");
        }
        return(0);
}
```

Sequence Points & Side Effects

https://stackoverflow.com/questions/16562266/c-side-effects-in-gcc-prefix-postfix-operator-and-precedence

- 1) If a side effect on a scalar object is unsequenced relative to another side effect on the same scalar object, the behavior is undefined.
  - i = ++i + i++; // undefined behavior
- The order of evaluation of parameters to a function call is also not specified
- **`int ans = i++,a+i;`** /* comma makes this well defined */
  /* (but not the best way to code it) */