CS 240 Programming in C

Flow of Control, Practice2

November 6, 2019
Statements and Blocks

- Recall that an expression is a combination of values, constants, variables, operators, and functions that evaluates to another value.
- An expression becomes a statement when it is followed by a semicolon. For example, `x = 0;`
- Curly braces are used to group declarations and statements into a compound statement, or block:
  ```
  { 
    x = 0;
    y = 1;
  }
  ```
- Note that the closing brace is not followed by a semicolon.
- Syntactically, the grouped statements are equivalent to a single statement.
If Statements

- Things to note:
  - The if condition is just testing a numeric value
  - We can use a shortcut in this test:
    - if (expression) same as if (expression != 0)
    - if (!expression) same as if (expression == 0)
Things to note:
- Can have as many as you want
- They are evaluated in order
- If the condition evaluates to true for one, its statement is executed, and we don’t look at the rest
- An else at the end is equivalent to "none of the above"
Switch Statements

- Another way to do multi-way decisions
  ```java
  switch (expression) {
  case constant-expr1:
    statements
  case constant-expr2:
    statements
  default:
    statements
  }
  ```

- This will test whether expression matches each of the constant expressions and execute the corresponding statements if so
- The constant expressions must be integer-valued
- Execution will fall through a switch (which means goes to the next switch statement) unless you add `break` after statements. That is to say an end of one case statement is not an end of one switch unless it is the last case statement.
```c
int main(void) {
    int c, i, cntWhite, cntOther, cntDigit[10];
    cntWhite = cntOther = 0;
    for (i = 0; i < 10; i++) cntDigit[i] = 0;
    while ((c = getchar()) != EOF) {
        switch (c) {
        case '0': case '1': case '2': case '3': case '4':
                  case '5': case '6': case '7': case '8': case '9':
            cntDigit[c - '0']++;
            break;
        case ' ': case '\n': case '\t':
            cntWhite++;
            break;
        default:
            cntOther++;
            break;
        }
    }
    printf("digits =");
    for (i = 0; i < 10; i++) printf(" %d", cntDigit[i]);
    printf(" , white space = %d, other = %d
", cntWhite, cntOther);
    return 0;
}
```

Haoyu Wang
If-Else vs. Switch

- Suppose we need to test the value of a status variable, and there are 20 different values.
- With `if`-else, we test `(status == 1)`, then `(status == 2)`, etc.
- By the time we reach 20, we have tested 19 times.
- With `switch`, it is usually compiled into assembly as a *jump table*.
- An array of `goto` instructions subscripted by the value of `status`.
- If `status` is 20, we look up the `goto` at address 20 in the table.
- This way we only execute that one `goto`.
- Good practice is to always use `break`.
- Falling through can be useful, but you should be careful with it as it may create unintended behavior if the program is modified later.
Loops

- These are equivalent

```plaintext
for (expr1; expr2; expr3)
  statement;

while (expr2) {
  statement;
  expr3;
}
```

- Note that any part of a for loop can be left out
  ```plaintext
  for(init; loop-test; increment)
  ```

- If `init` is left out, you must initialize somehow
- If `increment` is left out, you must manage increment
- If `loop-test` is left out, you must break
Comma Operator

- Most often use is in the for-loop statements
- Pairs of expressions separated by a comma, are evaluated left-to-right
- Value of comma expression is the value of the rightmost comma-separated expression
- Example of using the comma operator in a for-loop:

```
rev = (char *)malloc(sizeof(char) * (strlen(str) + 1));
for(i = 0, j = strlen(str) - 1; i < strlen(str); i++, j--)
    rev[i] = str[j];
rev[i] = '\0';
```
do {
    statements;
} while (expression);

- Guaranteed to execute the statements at least once, regardless of whether expression is true or false
- Used infrequently
**Break and Continue**

- **break**
  - Allows departure from a loop
  - Can be used in for, while, and do loops (similar to its use in switch)
  - Allows you to exit the current loop
    - one level only; remember this when you use break in nested loops

- **continue**
  - Skips to the next iteration of the loop
  - It is used to selectively execute statements in a loop iteration
```c
#include <string.h>

int trim(char str[]) {
    int n;

    for(n = strlen(str) - 1; n >= 0; n--)
        if (str[n] != ' ' && str[n] != '	' && str[n] != '
')
            break;

    str[n + 1] = '\0';
    return n;
}
```