Tutorial 5: Selection and Iteration Structures

Learning Outcomes

- Practice developing programs that include complex flow control statements.
- Develop familiarity with expressions and operators.
- Develop familiarity with more advanced use cases of pre-processor directives in C programs.

Topics and References

- Input/output. See this page and this page for details about function `printf` and its format specifications, respectively. See this page and this page for details about function `scanf` and its format specifications, respectively. Chapter 15 of the text contains information about functions `printf` and `scanf`.
- Selection structures. See this page for details of the `if` statement and the `else` clause. Chapter 5 of the text contains information about selection statements.
- Iteration structures. See this page for details of the `while` statement, this page for details of the `for` statement, and this page for details of the `do ... while` statement. Chapter 4 of the text contains information for iteration statements.

Task 1: Programming Selection Structures

Create a directory called `tut05` as a container and work space for files related to this task.

Implement a function `cost` in source file `q.c` as follows: `cost` takes in no parameters and returns nothing. `cost` determines the cost of an automobile insurance premium, based on driver's age and the number of accidents that the driver has had. The basic insurance charge is $500. It is illegal for drivers younger than 16 years to drive and therefore such drivers cannot be insured. There is a surcharge of $50 if the driver is under 25 years of age and an additional surcharge for accidents:

# of accidents	Accident Surcharge
0	$0
1-2	$100
3-4	$225
5 or more	no insurance coverage

The function must read the driver's age and accident record from standard input. Using the above table, your program must print to standard output the calculated cost of insurance for this driver. A sample interaction with the program would look like this:

First of all, age and number of accidents are read in from the standard input device as follows:

```
25, 3
```

Then the output from the function to the standard output device is:

```
age = 25, number of accidents = 3, insurance cost = 725

```

Note the newline character at the end of the output indicated by the empty line 2 above. Here's a second example: Input of -

```
22,8
```

followed by the following output:

```
age = 22, number of accidents = 8, no insurance coverage!

```

The exercise here is to avoid using incorrect and incompatible data from the user that is then subsequently used to print strange amounts in a customer bill. Write down all the different ways in which the program can be supplied incorrect data. For each such specific scenario, alter your algorithm (if you don't have one, you're not adopting and applying good practices you learned in the previous weeks) to avoid using incorrect and incompatible data.

Use the input and output files to understand and implement the correct behavior expected from your program. Your program must be able to read the input file and generate output that matches the text in the corresponding output file.

In this exercise, code in `qdriver.c` is to call `cost`. To enable such code to do so, copy all the code below into `q.h` and fill in the needed code so that `qdriver.c` just needs to include `q.h` in order to call `cost`.

```c
#ifndef _Q_H_ // Lines 1, 2 & 7 form the header guards
#define _Q_H_

// @todo declaration of cost here
void rev(void); // Copy this too. It's for Task 2 that's coming later

#endif //_Q_H_

```

A directive `#ifndef` (*if not defined*) and its matching `#endif` delimit a fragment of code that will be included in a produced translation unit only if a given identifier has not yet been defined.

This code represents a very common pattern called **header guards** that lets C programmers avoid including the fragment twice and thus introducing duplicated definitions. An identifier's name is derived from the name of the file; in this case it is `_Q_H_` for the filename *q.h*. The idea is to have direct mapping from a filename to an identifier which is likely to be unique for the file.

By default, all identifiers are undefined to the preprocessor. If the current header file has not yet been included, this means the identifier `_Q_H_` has not been defined, and the preprocessor will include the fragment between `#ifndef` and `#endif`; by doing so it will also define the identifier with `#define` since `#define` lies between `#ifndef` and `#endif`.

Later, if the file is included again, the directive `#ifndef` would find `_Q_H_` already defined and so omit the code between `#ifndef` and `#endif`. The single-line comment next to `#endif` in line 7 is there to inform human readers about the macro related to this condition directive. This is because header files can be very large and the `#endif` directive may be far removed from its corresponding `#ifndef`.

Testing

Notice that `q.h` has a declaration for `rev`. This is because `qdriver.c` calls `rev` which is to be completed as part of Task 2. For now, in order to compile `q.c` with `qdriver.c`, create a stub function for `rev` in `q.c` which is a function definition that has an empty body consisting of braces with no C statements inside i.e. a body that does nothing.

Compile `q.c` with the full suite of `gcc` flags:

```
gcc -std=c11 -Wall -Werror -Wextra -Wconversion -Wstrict-prototypes -
pedantic-errors -c q.c -o q.o
```

You will not be editing `qdriver.c`. Instead, you'll be using the calls from function `main` defined in `qdriver.c` to test your definitions in source file `q.c`. Separately compile source file `qdriver.c` into an object file `qdriver.o`:

```
gcc -std=c11 -Wall -Werror -Wextra -Wconversion -Wstrict-prototypes -
pedantic-errors -c qdriver.c -o qdriver.o
```

After successfully compiling the two source files, the next step is to link the two object files `q.o` and `qdriver.o` into an executable `q.out`:

```
gcc qdriver.o q.o -o q.out
```

Run the executable and manually key in the example inputs above and check the output. Then run the program using `input.txt` as input and direct the results to `your-output1.txt`:

```
./q.out < input.txt > your-output1.txt
```

Compare your output with the expected like this:

```
diff -y --strip-trailing-cr --suppress-common-lines your-output1.txt
expected-output1.txt
```

If `diff` reports differences, you must go back and amend `q.c` to remove those differences.

Iteration Structures

Iteration structures are used to implement repetitive structures. C/C++ contain three different iteration structures - the `while`, the `do...while`, and the `for` loops. In addition, C/C++ also provide two additional statements to modify the behavior of iteration statements - the `break` statement and the `continue` statement.

`while` statement

The general form of a `while` loop is as follows:

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

The `expression` is evaluated before the statements `statement` within the loop are executed. If `expression` evaluates to a `0` value, the loop statements are skipped, and execution continues with the statement following the `while` statement.

The following program uses a `while` loop to generate a conversion table for converting degrees to radians:

```
/*!
@file     convert-while.c
@author   Nicolas Pepe  (nicolas.pepe@digipen.edu)
@course   CS 120
@section  ?
@date     02/02/2020
@brief    This program prints a degree-to-radian table using a
          while iteration statement.
*//*_____________________________________________________________*/

#include <stdio.h>
#define PI 3.141593

int main(void) {
  // declare and initialize variables
  int degrees = 0;
  double radians;

  // print degrees and radians in a loop
  printf("%10s%12s\n", "Degrees", "Radians");
  while (degrees < 360) {
    radians = ((double)degrees * PI)/180.0;
    printf("%8d%13.2f\n", degrees, radians);
    degrees += 10;
  }

  return 0;
}

```

The first few lines of output from the program look like this:

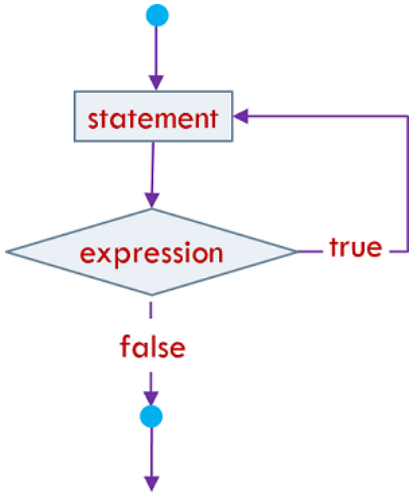
```
   Degrees     Radians
         0         0.00
        10         0.17
        20         0.35
        30         0.52
```

`do...while` statement

The general form of the `do...while` loop is as follows:

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

The execution flow of a `do...while` loop is illustrated in the following picture:



The `statement` executes first, and then `expression` is evaluated. If `expression` evaluates to a non-zero value (or, *true*), `statement` executes again. As long as `expression` in a `do...while` evaluates to a non-zero value (that is, *true*), the `statement` executes.

A `while` loop is called ***pretest*** iteration - the iteration condition is evaluated before executing the body of the loop. This means that because of the pretest condition, the while loop may never have its body executed. On the other hand, the `do...while` loop is a ***posttest*** loop. Since the `do...while` loop is a posttest loop, its body will be executed at least once. Consider the following `while` loop:

```
// assume i is an int
i = 26;
while (i <= 25) {
  printf("%d ", i);
  i += 5;
}
```

The `while` loop prints nothing because the expression `i <= 25` evaluates to `0` and therefore the body of the loop will never execute. Now, consider the following `do...while` loop:

```
i = 26;
do {
  printf("%d ", i);
  i += 5;
} while (i <= 25);
```

The `do...while` loop prints the integral value `26` and also changes the value of `i` to `31`.

What is the main purpose of the `do...while` loop? It's most useful manifestation is for input validation. Suppose that a program prompts a user to enter a character whose value is either `r` for Regular or `p` for Premium. If the user enters a character that is neither `r` nor `p`, the user should be prompted to re-enter the character. The `do...while` loop can implement this objective.

The following program prints the degree-to-radian conversion table using a `do-while` loop instead of a `while` loop:

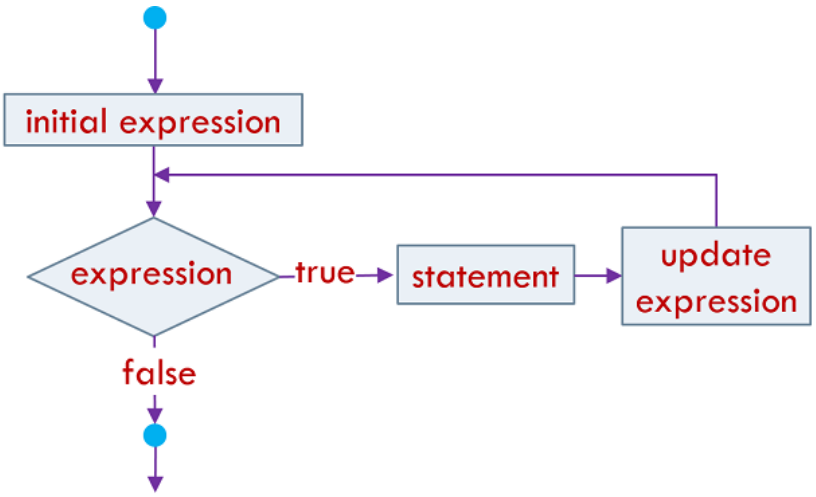
```c
/*!
@file     convert-dowhile.c
@author   Prasanna Ghali (pghali)
@course   CS 120
@section  ALL
@date     08/04/2016
@brief    This program prints a degree-to-radian table using a
          do-while iteration statement.
*//*_____________________________________________________________*/

#include <stdio.h>
#define PI 3.141593

int main(void) {
  // declare and initialize variables
  int degrees = 0;
  double radians;

  // print degrees and radians in a loop
  printf("%10s%12s\n", "Degrees", "Radians");
  do {
    radians = ((double)degrees * PI)/180.0;
    printf("%8d%13.2f\n", degrees, radians);
    degrees += 10;
  } while (degrees < 360);

  return 0;
}
```

`for` statement

Many programs require iterations that are based on the value of a variable that increments (or decrements) by the same amount each iteration. When the variable reaches a specified value, we'll want to exit the iteration statement. This type of iteration structure can be implemented as a `while` loop, but it can also be easily implemented with the `for` loop. The general form of the `for` loop is as follows:

```c
for (initial expression; expression; update expression) {
  statements
}
```

The execution flow of a `for` loop structure is illustrated in following picture:



The `initial expression` is used to initialize a loop-control variable, `expression` specifies the condition that must be true to continue the loop repetition, and `update expression` specifies the modification to the loop-control variable.

If you're thinking that the `for` loop shown above looks similar to the `while` loop, then you're on the right track. Every `for` loop can be easily rewritten as a `while` loop. For example, the above `for` loop can be rewritten using the following `while` loop:

```
initial expression;
while (expression) {
  statement
  update expression;
}
```

If you want to execute a loop 15 times, with the value of the variable `i` going from 0 to 14 in increments of 1, you could use the following `for` loop:

```
int i;
// other code here
for (i = 0; i < 15; ++i) {
  statement
}
```

The `update expression` which is written as `++i` could have been written as `i++`, or as `i += 1`, or as `i = i+1`.

If you want to execute a loop with the value of the variable `j` going from 20 to 0 in increments of -2, you could use this `for` loop:

```
int j;
// other code here
for (j = 20; j >= 0; j -= 2) {
  statement
}
```

There's a neat trick for computing the number of times that a `for` loop will be executed:

$$\left\lfloor \frac{final\ value - initial\ value}{increment} \right\rfloor + 1$$

If this value is negative, the loop is not executed. If a `for` loop has the structure:

```
int k;
// other code here
for (k = 12; k <= 89; k += 4) {
  statement
}
```

it would be executed the following number of times:

$$\left\lfloor \frac{89-12}{4} \right\rfloor + 1 = \left\lfloor \frac{77}{4} \right\rfloor + 1 = 20$$

The following program prints the degree-to-radian conversion table using a `for` loop. Note that the pseudocode for the `while` and `for` loops are identical for the conversion table problem.

```
/*!
@file     convert-dowhile.c
@author   Prasanna Ghali (pghali)
@course   CS 120
@section  ALL
@date     08/04/2016
@brief    This program prints a degree-to-radian table using a
          for iteration statement.
*//*_____________________________________________________________*/

#include <stdio.h>
#define PI 3.141593

int main(void) {
  // declare and initialize variables
  int degrees;
  double radians;

  // print degrees and radians in a loop
  printf("%10s%12s\n", "Degrees", "Radians");
  for (degrees = 0; degrees < 360; degrees += 10) {
    radians = ((double)degrees * PI)/180.0;
    printf("%8d%13.2f\n", degrees, radians);
  }

  return 0;
}
```

Task 2: Programming an Iteration Structure

Preferably using a `do...while` iteration statement, implement the function `rev` in `q.c` that prints an integer backward. The function must read the number from standard input and print the reversed number to standard output.

If the value `1234` is entered, the function prints the following:

```
4321

```

Note the newline at the end of `4321`. If `1234567890` is entered, it prints:

```
0987654321

```

As for `-1234`, the output is:

```
-4321

```

I'll let you think about how to use integer operators that we have encountered so far to extract each digit of an integer. When you have implemented `rev`, compile an object file from `q.c`:

```
gcc -Wall -Werror -Wextra -Wconversion -Wstrict-prototypes -pedantic-errors -std=c11 -c -o q.o q.c
```

If the compilation resulted in any errors, resolve them and try again. Once `q.c` compiles without error, that would mean both translation units, i.e. the two `.c` files, are compilable. This would mean you can also compile and link them together to create an executable:

```
gcc -Wall -Werror -Wextra -Wconversion -Wstrict-prototypes -pedantic-errors -std=c11 -o q.out qdriver.c q.c
```

With `cost` and `rev` implemented, your program must be able to read `input.txt` and generate output that ***exactly*** matches the text in `expected-output2.txt`. Check with the following:

```
./q.out < input.txt > your-output2.txt
```

Followed by:

```
diff -y --strip-trailing-cr --suppress-common-lines your-output2.txt expected-output2.txt
```

If there are differences reported by `diff` go back and resolve them by fixing `q.c`.

Add file-level documentation

In **Laboratory exercise 2** you have learnt that every source code file you submit for grading must start with a file-level documentation header. Open the specification of that exercise and use the template of a header provided there in `q.h` and `q.c`. Replace `@todo` and the rest of each line with your information. When you edit the file later, remember to check if the information is up-to-date.

The file-level documentation should precede any content of a source file, including the header guards appropriate for all header files.

Add function-level documentation

Since all functions in this exercise are well defined, you are ready to write their function-level documentation.

In doing **Assignment 2** you would have learnt that each function in every source code file you submit for grading must be preceded by a function-level documentation header that explains its purpose, inputs, outputs, and side effects, and that lists any special considerations. Open the specification of that exercise and review the example of function header documentation provided there; add relevant function-level documentation for all functions in `q.h` and `q.c`.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit files `q.h` and `q.c`.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
    - F grade if your submission doesn't compile with the full suite of `gcc` options.
    - F grade if your submission doesn't link to create an executable.
    - Proportional grade if the program's output doesn't fully match the correct output.
    - A+ grade if the submission's output fully matches the correct output.
    - Possible deduction of one letter grade for each missing file header documentation block. Every submitted file must have one file-level documentation block and each function must have a function-level documentation block. A teaching assistant may physically read submitted files to ensure that these documentation blocks are authored correctly. Each missing block may result in a deduction of a letter grade. For example, if the automatic grader gave your submission an A+ grade and one documentation block is missing, your grade may be later reduced from A+ to B+. Another example: If the automatic grader gave your submission a C grade and two documentation blocks are missing, your grade may be later reduced from C to F.