day-3-control-flow

September 6, 2023

1 Day 3: Control flow

1.0.1 Topics

- control flow
  - conditionals (if/switch/ternary)
  - loops (while/do/for)
  - jumps (break/continue/goto)

1. Conditionals 1.1. `if-else`

Write a short C program in which you test whether an input unsigned integer $n \in \mathbb{N}$ is even ($n \mod 2 = 0$) or odd ($n \mod 2 = 1$). If it is even print $n/2$; if it is odd, print $3n + 1$. You may recognize this as the Collatz map.

[ ]:
```c
//// collatz-map.c

int main(void) {
    return 0;
}
```

1.2. `switch-case`

Write a short C program which takes user input for the day, month, and year as unsigned integer values, and then prints the date these correspond to in English format. That is, if the inputs are `12`, `1`, `1968` then your program should print `January 12, 1968`.

[ ]:
```c
//// print-the-date.c

int main(void) {
    return 0;
}
```

1.3. `?...:...` (the *ternary* operator)

The ternary operator syntax is a convenient shorthand for `if ... else` statements, but is an **expression**, and can thus appear on the right-hand side of an assignment. This expressivity means that it can affect assignment easily. We will use the ternary operator to check whether a user-input year is a leap year, by checking whether it evenly divides 4, 100, and/or 400. Translate the `if ... else` conditional structures in `ternary-tests.c` below into ternary `? ... : ...` ones. Try to

make the actual detection of the leap year decision happen on a single line, and reduce the number of print statements so each is unique.

```c
//// ternary-tests.c
#include <stdio.h>

int main(){
    int year;
    printf("Please input a year:\n");
    scanf("%d", &year);
    if (year % 4==0){
        if (year % 100!=0){
            printf("\t%d is a leap year!\n", year);
        }
        else{
            if (year % 400==0){
                printf("\t%d is a leap year!\n", year);
            }
            else{
                printf("\t%d is not a leap year!\n", year);
            }
        }
    }
    else{
        printf("\t%d is not a leap year.\n", year);
    }
    return 0;
}
```

[ ]:
```c
//// ternary-tests.c
#include <stdio.h>

int main(void){
        int year;
        printf("Please input a year:\n");
        scanf("%d", &year);
    // your solution here!
    return 0;
}
```

2. Loops 2.1. `while`

Using your Collatz map code from above, implement a checker for the Collatz conjecture for a user input `unsigned int`. Correctly track the iteration and include that in a print out in the loop which includes the iteration, current iterate, and the mapping of the current iterate.

Instead of the usual

$$f(n) = \begin{cases} n/2, & n \mod 2 = 0 \\ 3n+1, & n \mod 2 = 1 \end{cases}$$

You may want to use

$$f(n) = \begin{cases} n/2, & n \mod 2 = 0 \\ (3n+1)/2, & n \mod 2 = 1 \end{cases}$$

so that the running time of your program is lessened.

[ ]:
```c
//// collatz-checker.c

int main(void) {
    return 0;
}
```

2.2. `do ... while`

The `do ... while` loop is much less common than `for` or `while` loops. These loops guarantee that the body of the loop is run at least once, which can be helpful for all sorts of dynamically terminating algorithms. Implement a cumulative sum program which repeatedly takes an `int` as user input and prints the running total, until and unless the user inputs `0`.

[ ]:
```c
//// running-total.c

int main(void) {
    return 0;
}
```

2.3. `for`

The `for` loop is the true workhorse of C programming, and probably the most used in computational science applications. In this exercise we will look at the logistic map and explore its dynamics. Write a short program which allocates two `double` arrays of length $M = 500$, `double x[M], r[M]`. Assign the initial values $r_i = 0 + 4i/500$ and $x_i = 0.5$. For each index i, evaluate the logistic map for the i-th value of x, `x[i]`, using the i-th value of r, `r[i]`, for $N = 100$ iterations:

$$x_{i,n+1} \leftarrow r_i \cdot x_{i,n}(1 - x_{i,n}).$$

Using `fprintf` print your results to `stdout` first. Once this is working, extend your program to write your results to a file so that you can plot your data using your faviourite graph plotting software.

[ ]:
```c
//// logistic.c

int main(void) {
    return 0;
}
```

Write a program which re-implements the Collatz checker from earlier but, instead of asking for user input, exhaustively find the smallest input with the longest Collatz sequence among all possible values of `unsigned short int`. You might find `limits.h` useful here!

[ ]:
```c
//// collatz-uint-short.c

int main(void) {
    return 0;
}
```

3. Jumps 3.1. `continue` & `break`

In this exercise, we want to write a variant of FizzBuzz - a drinking game for teaching children integer factorization - mixed with a bit of procedural poetry.

First, you should create a string with enough characters so that it will be eerie to see some of it missing when printed out by the terminal. Maybe you could use a line from a favourite book or movie.

Your program should iterate over $n \in [1, L]$ where L is the length of your string, and then for each $m \in [1, n/2]$:

- If n is evenly divisible by m but not $m+1$, then you should print out the $n-1$-th character in your string.
- If n is evenly divisible by m and $m+1$, then you should print the $n/(m+1)$-th character before skipping to the next inner iteration.
- If n is evenly divisible by m and $n \mod (m+1) = m$, then you should skip to the next outer loop iteration.
- If n is not evenly divisible by m but is evenly divisible by $m+1$, then you should print your whole string.

After checking all these conditions, you should print the m-th character of the string.

[ ]:
```c
//// continue-break.c

int main(void) {
    return 0;
}
```

3.2. `goto`

Write a variant of the exhaustive Collatz checker from earlier to search for the smallest `unsigned short int` which results in a sequence longer than some user-specified value. Your program should stop searching as soon as the smallest such result is found. Try this in 2 ways:

- Using `break` (*hint*: remember that `break` only stops the inner-most looping construct).
- Using a single, well-placed `goto` statement.

The use of `goto` is often considered to be a ``code smell'', but there are a few legitimate use-cases, such as breaking out of nested loops. Which approach do you think results in cleaner code in this case?

[ ]:
```c
//// collatz-first-sequence-longer-than.c

int main(void) {
    return 0;
}
```

Let's now write a program which re-implements the original Collatz checker from earlier using `goto`, and conditionals **without loops**.

[ ]:
```c
//// collatz-checker-using-goto.c

int main(void) {
    return 0;
}
```