C track: assignment 2

Goals

In this assignment you will write a more substantial program and learn some new language constructs.

Language concepts covered this week

- operators
- functions
- function prototypes
- more on loops
- comments

Other concepts covered this week

- the `make` program and Makefiles
- I/O redirection (unix)
- test scripts

Reading

Read this page to familiarize yourself with the `make` program and Makefiles.

Also take a look at our C style guide. Starting from this lab, I'll be more picky about style issues.

Operators and operator precedence in C

An "operator" is a character or character sequence that has a special syntactic meaning to the C compiler. Most operators are binary, which means that they are found sandwiched between two values. An example is the addition operator, `+`, which is found in expressions such as `1 + 2`. Some operators are unary, such as the bitwise-NOT operator (`~`). One operator is actually trinary (the dreaded `? :` operator), but we won't discuss it here. Compared to most languages, C has a very large number of operators and a correspondingly large number of operator precedence levels (15 of them to be precise; see table 2-1 in K&R if you're curious). Operator precedence levels determine how to interpret expressions with multiple operators. For instance,

```
a = b + c * d + e;
```

is interpreted as being

```
a = b + (c * d) + e;
```

because multiplication has higher precedence than addition. If we want it to be interpreted differently, we need to use parentheses, *e.g.*

```
a = (b + c) * (d + e);
```

Note that the = sign is also an operator (the assignment operator). It has very low precedence, so we don't need to use parentheses around the arithmetic expression to the right of the = sign. Also note that (confusingly) equality testing uses the == operator, not the = operator.

We **do not** want you to memorize the operator precedence table! Instead, simply use these three rules:

1. Multiplication and division have precedence over addition and subtraction;
2. All assignment operators (except for ++ and --) have extremely low precedence;
3. Put parentheses around everything else where there is any possibility of confusion.

Also note that C has a lot of shortcut assignment operators:

- Operators of the form `op=` *e.g.* `+=`, `-=`, `*=`, `/=`, `%=`, etc. These all have this meaning:

  ```
  x op= y;
  ```

  means:

  ```
  x = x op y;
  ```

  for some operator "`op`".

- The ++ and -- increment/decrement operators.

Used judiciously, they often result in more concise and understandable code. An annoying fact is that the ++ and -- operators have a very *high* precedence, whereas all the other assignment operators have the same precedence as = does.

Program to write

You will write a program called `easter` that will compute the day of the year on which Easter falls, given the year.

Description of the algorithm

This algorithm is taken from Donald Knuth's famous book The Art of Computer Programming (see the references below).

```
GIVEN:   Y: the year for which the date of Easter is to be determined.
FIND:    The date (month and day) of Easter

STEP E1: Set G to (Y mod 19) + 1.
         [G is the "golden year" in the 19-year Metonic cycle.]
STEP E2: Set C to (Y / 100) + 1. [C is the century]
STEP E3: Set X to (3C / 4) - 12. [X is the skipped leap years.]
         Set Z to ((8C + 5) / 25) - 5.
         [Z is a correction factor for the moon's orbit.]
STEP E4: Set D to (5Y / 4) - X - 10.
         [March ((-D) mod 7 + 7) is a Sunday.]
STEP E5: Set E to (11G + 20 + Z - X) mod 30.
         If E is 25 and G is greater than 11 or if E is 24,
         increment E.
         [E is the "epact" which specifies when a full moon occurs.]
STEP E6: Set N to 44 - E.  [March N is a "calendar full moon".]
         If N is less than 21 then add 30 to N.
STEP E7: Set N to N + 7 - ((D + N) mod 7).
         [N is a Sunday after full moon.]
STEP E8: If N > 31 the date is APRIL (N - 31),
         otherwise the date is MARCH N.
```

Note: all divisions in this algorithm are integer divisions, which means that any fractional remainders are thrown away. Also, the comment "March ((-D) mod 7 + 7) is a Sunday" is technically only true for years after 1752, because there was an 11-day correction applied to the calendar in September of 1752. You don't need to mention this in your comments, since it doesn't affect the Easter computation.

Note 2: We will be adding another step to make the algorithm work nicely with our C program; see the description of the program below for more details.

Note 3: Just because the great Don Knuth wrote this algorithm this way doesn't mean that it's written in a nice or easy-to-understand way. In particular, the use of single characters as variable names is usually a very bad idea (because single characters don't have any meaning to the person reading the code), and we don't want you to do that in this program. Knuth was trying to describe the algorithm in as short a space as possible; you don't have that restriction.

Explanation of the algorithm

Ever wonder what those monks did during the Dark Ages, all secluded away in their distant mountaintop monasteries and things? Well, it turns out that they were busy calculating the date of Easter. See, even back then, there wasn't much point in spending any effort on calculating the dates of holidays like Christmas, which as everyone knows, is on the same day each year. That also went for holidays which have become a tad more obscure, like Assumption (August 15th).

But the trouble with Easter is that it has to fall on Sunday. I mean, if you don't have that,

all the other non-fixed holidays get all screwed up. Who ever heard of having Ash Wednesday on a Saturday, or Good Friday on Thursday? If the Christian church had gone and made a foolish mistake like that, they'd have been the laughingstock of all the other major religions everywhere.

So the Church leaders hemmed and hawwed and finally defined Easter to fall on the first Sunday after the first full moon after the vernal equinox.

I guess that edict must not have been too well-received, or something, because they then went on to define the vernal equinox as March 21st, which simplified matters quite a bit, since the astronomers of the time weren't really sure that they were up to the task of finding the date of the real vernal equinox for any given year other than the current one, and often not even that. So far so good.

The tricky part all comes from this business about the full moon. The astronomers of the time weren't too great at predicting that either, though usually they could get it right to within a reasonable amount, if you didn't want a prediction that was too far into the future. Since the Church really kinda needed to be able to predict the date of Easter more than a few days in advance, it went with the best full-moon-prediction algorithm available, and defined "first full moon after the vernal equinox" in terms of that. This is called the Paschal Full Moon, and it's where all the wacky epacts and Metonic cycles come from.

So what's a Metonic Cycle?

A Metonic cycle is 19 years.

The reason for the number 19 is the following, little-known fact: if you look up in the sky on January 1 and see a full moon, then look again on the same day precisely 19 years later, you'll see another full moon. In the meantime, there will have been 234 other full moons, but none of them will have occurred on January 1st.

What the ancient astronomers didn't realize, and what makes the formula slightly inaccurate, is that the moon only really goes around the earth about 234.997 times in 19 years, instead of exactly 235 times. Still, it's pretty close -- and without computers, or even slide rules, or even pencils, you were happy enough to use that nice, convenient 19-year figure, and not worry too much about some 0.003-cycle inaccuracy that you didn't really have the time or instruments to measure correctly anyway.

Okay, how about this Golden Number business then?

It's just a name people used for how many years into the Metonic cycle you were. Say you're walking down the street in Medieval Europe, and someone asks you what the Golden Number was. Just think back to when the last 19-year Metonic cycle started, and start counting from there. If this is the first year of the cycle, it means that the Golden Number is 1; if it's the 5th, the Golden number is 5; and so on.

Okay, so what's this Epact thing?

In the Gregorian calendar, the Epact is just the age of the moon at the beginning of the year. No, the age of the moon is not five billion years -- not here, anyway. Back in those days, when you talked about the age of the moon, you meant the number of days since the moon was "new". So if there was a new moon on January 1st of this year, the Epact is zero (because the moon is new, *i.e.* zero days "old"); if the moon was new three days before, the Epact is three; and so on.

When Easter was first introduced, the calculation for the Epact was very simple -- since the phases of the moon repeated themselves every 19 years, or close enough, the Epact was really easy to calculate from the Golden Number. Of course, this was the same calendar system that had one leap year every 4 years, which turned out to be too many, so the farmers ended up planting the fields at the wrong times, and life just started to suck.

Pope Gregory Makes Things More Complicated

You probably already know about the changes Pope Gregory XIII made in 1582 with respect to leap years. No more of this "one leap year every four years" business like that Julius guy said. Nowadays, you get one leap year every four years *unless* the current year is a multiple of 100, in which case you don't -- ***unless*** the current year is *also* a multiple of 400, in which case you do anyway. That's why 2000 was a leap year, even though 1900 wasn't (I'm sure many of you were bothered by this at the turn of the millenium).

Well, it turns out that the *other* thing Pope Gregory did, while he was at it, was to fix this Metonic Cycle-based Easter formula which, quite frankly, had a few bugs in it -- like the fact that Easter kept moving around, bumping into other holidays, occuring at the wrong time of year, and generally making a nuisance of itself.

Unfortunately, Pope Gregory had not taken CS 11. So instead of throwing out the old, poorly-designed code and building a new design from scratch, he sort of patched up the old version of the program (this is common even in modern times). While he was at it, he changed the definition of Epact slightly. Don't worry about it, though -- the definition above is the new, correct, Gregorian version.

This is why you'll see Knuth calculating the Epact in terms of the Golden Number, and then applying a "correction" of sorts afterwards: Gregory defined the Epact, and therefore Easter, in terms of the old definition with the Metonic cycles in it. Knuth is just the messenger here.

So what is this thing with "Z" and the moon's orbit?

It's just the "correction" factor which the Pope introduced (and Knuth later simplified) to account for the fact that the moon doesn't really orbit the earth exactly 235 times in 19 years. It's analogous to the "correction factor" he introduced in the leap years -- the new

formula is based on the old one, is reasonably simple for people who don't like fractions, is also kind of arbitrary in some sense, and comes out much closer to reality, but still isn't perfect.

What about all the rest of that stuff?

Ah, well, you wouldn't want me to make this too easy, would you? My hope is that, after this brief introduction, that code up there will not seem quite so mysterious, and that you may, in fact, be able to figure out, if not exactly what's going on, at least most of the stuff that's happening in there.

Description of the program

Write a program that reads a series of years from a text file and prints out the date of Easter on all those years, as follows.

- **Running the program**

  When the program is written, use Unix input/output redirection to handle input from and output to files (if you don't know about this, here is a decent tutorial). In other words, invoke the program like this:

  ```
  % easter  < infile  > outfile
  ```

  where `%` is the Unix prompt (so you don't type that in). **Note:** this is Unix syntax, **not** C syntax! The `< infile` part means to take the input from the file called `infile` instead of from the keyboard, and the `> outfile` part means to send the output to the file called `outfile` instead of printing it to the terminal. See below for more details on this.

  Note: You can't leave out the < and > characters *i.e.* don't do this:

  ```
  easter infile outfile
  ```

  `infile` is a file containing a list of years (one per line) *e.g.*

  ```
  1994
  1995
  1998
  ```

  and `outfile` will become a list of year/date pairs, *e.g.*

  ```
  1994 - April 3
  1995 - April 16
  1998 - April 12
  ```

  Handling input from files and output to files directly from your C program is possible, but it's a little bit more complicated, so we won't bother with it now (it involves functions like `fopen()`, `fclose()` and `fscanf()`; look them up if you're

curious). Instead, you just have to read from standard input (*i.e.* the terminal; use `scanf()` like in the previous lab) and write to standard output (using `printf()`). Unix operating systems (including Linux, which is what the CS cluster machines are running) will convert this to reading from a file and writing to a file if you use the < and > symbols in the command line as we showed above:

```
% easter  < infile  > outfile
```

Technically, what this does is bind standard input to the file `infile` and bind standard output to the file `outfile` just for this one invocation of the `easter` program, so that when in your program you read from standard input (using `scanf()`) you're really reading from `infile`, and when you write to standard output (using `printf()`) you're really writing to `outfile`. If you think this is kind of cool, then you're right.

- **Easter computation**

  The program should include a function called `calculate_Easter_date` (yes, that exact name) which takes an integer argument (the year) and returns an integer representing the date. The month should be indicated by the sign of the integer return value: negative means March and positive means April. The absolute value of the integer represents the day of the month. So April 10 would be represented as the integer 10, while March 23 would be represented as the integer -23. **This is not part of the Knuth algorithm!** You have to convert from the value that Knuth's algorithm gives you to the value in this representation (which is quite easy).

  [**NOTE**: Returning the date this way is an egregious hack. There are much better ways to return multiple values from a C function, which will be described in later labs.]

  The allowable years are in the range 1582 to 39999; if the input is outside of this range, the `calculate_Easter_date` function should return 0. When the main program sees this return value, it should print an error message to `stderr` (NOT to `stdout`; use `fprintf` instead of `printf` for this), and then continue with the loop.

  NOTE: Don't print the error message inside the `calculate_Easter_date` function; do it inside the `main` function. This is good design; the `calculate_Easter_date` function should only be concerned with calculating the Easter date; printing error messages are not its responsibility. In general, you should design functions to do one thing and only one thing; it'll make your programs much more elegant and much easier to debug.

- **Input/output**

  In the `main()` function, use a call to `scanf` to read in each the input line from `stdin`. Note that to read an integer value using `scanf`, you need to use the "%d" format string (where "d" means "decimal"). Store the return value of `scanf` in a

variable (yes, `scanf` does return a value, but it isn't used very often). This return value will be equal to the integer constant `EOF` ("end of file") when there is no more input. `EOF` is defined in the header file `<stdio.h>`, just like `printf` and `scanf`. You will find the `break` statement (K&R pp. 64-65) to be useful in your loop. Use a `while` loop that loops forever *e.g.* `while (1) { ... }` until an `EOF` is encountered, and then `break` out of the loop. The `main()` function should call the `calculate_Easter_date()` function for each line of input. If `calculate_Easter_date()` returns 0 (because of a range error), print an error message to `stderr` and keep going.

Make sure you have declared function prototypes at the top of your file before you define the functions. Although this isn't strictly necessary here, it's a good habit to get into. Prototypes allow you to reference functions before they are defined, which allows you to program without having to worry about what order your functions are defined in. You do not have to write a prototype for the `main` function.

- **Commenting**

  Comment your code liberally, especially the Easter algorithm itself. Very few programmers write too many comments; most write way too few. Remember, your task is to write a program which (a) does what it's supposed to, and (b) is clearly understandable. You are free to copy Knuth's algorithm verbatim if you like, but make sure you add comments explaining what each step of the algorithm does. Also, your version of the algorithm should contain better variable names than the ones Knuth uses. If you use one-letter variable names like Knuth does, you'll have to redo the program. Instead, use variable names that are words or phrases that are descriptive of the meaning of the variable.

  For functions, **put a comment before the function that states what the function does, what the arguments mean, and what the return value means.** These are the most important kinds of comments you'll ever write, because they are what will allow other people to use your code.

  Also, put a comment at the top of the file explaining what the program does as a whole.

  We will be grading your program not only on how well it performs its task, but on how easy it is to read and understand. Make sure you keep this in mind as you do this lab. And don't think this is only an exercise for this lab -- future labs have to have the same standard of commenting.

Testing the program

We are supplying you with a simple `Makefile`. Download it into your lab2 directory as a file called "`Makefile`" (it should be called that by default; just don't change the name). Running "`make`" will create the `easter` program. Running "`make test`" will run a

simple test of the program and report whether it is correct or not with respect to the inputs. This is an example of a "test script". Test scripts are critically important in producing correctly-working code. Some programmers even advocate writing test scripts and test cases for functions before writing the actual code to be tested. Running "`make check`" will run the style checker on your code.

Other things to do

Write a `clean` target for the Makefile. This target will remove the `easter` program, all object files, and all files generated by the program when you type "`make clean`". It must **not** remove the input file used to test the program, your source code file, the test script, or the correct output file. **Hint:** use the Unix command "`rm -f`" (*i.e.* the `rm` program with the `-f` optional argument) to remove your files. Normally, `rm` complains if you try to remove a nonexistent file, but with the `-f` optional argument it won't. Note also that "`rm -f`" can take multiple filename arguments.

Try the program on an input file that intentionally contains years that are outside the correct range. Send the output to a file as usual. The error messages should be printed on the terminal, not put into the file. This shows you the difference between printing to `stdout` (which `printf()` does) and printing to `stderr` (which `fprintf()` does if you tell it to).

Supporting files

Make sure you download all of these files into your `lab2` directory (except possibly for the style checker, which ideally you should already have put into your `~/bin` directory).

- The Makefile.

  NOTE! Be sure that all the command lines in the Makefile start with tabs, or they will not work. The easy way to do this is not to try to copy and paste from the browser window. Instead, use the "Save Page As" function of your browser (which is probably under the File menu).

- The test script.

  In order to get this to work, you have to do "`chmod +x run_test`" after downloading this file, in order to make it executable. Make sure that the resulting file name is "`run_test`" and not (for instance) "`run_test.txt`", or it won't work!.

- The input file, for testing.

- The correct output file, for testing.

- The style checker.

To hand in

The program `easter.c` and the completed `Makefile`. We will run the program to see if it passes the test script.

For extra credit...

Have your program use Zeller's Congruence to verify that the date it is printing really does fall on a Sunday. Use `assert` to signal an error in case it doesn't. Get the definition of Zeller's Congruence by doing an internet search. Type "`man assert`" to learn more about the `assert` macro. Assert is a very valuable and under-appreciated debugging aid, which we will meet again in later labs.

References

- K&R, chapters 2, 3, and 4.
- Donald Knuth, The Art of Computer Programming, vol. 1: Fundamental Algorithms.

  The Art of Computer Programming is a three-volume set (although more volumes are being written) which is a definitive treatment on computer algorithms written by Donald Knuth. The books are usually referred to simply as "Knuth vol. 1", etc. They are extremely dense and not really suitable for beginners, but they are good if you need to look up an algorithm and learn more about it. Knuth virtually invented the field of mathematical analysis of computer algorithms, and is still going strong.
