CP Lab 2: Programs for simple arithmetic problems

Instructions

The purpose of this Lab is to guide you through a series of simple programming problems, using the basic types of `int` and `double`; the basic arithmetic operations; and also the `if`-statement construct.

All Labs for CP will take place on the DICE machines in AT5.05 (Thursday and Tuesday) or AT6.06 (Monday). For working alone, you may use any of the DICE labs in AT, as long as they are not booked out by a class. You have 24-hour access to AT levels 3,4,5: if your card/pin does not allow you in, use the ITO contact form on our course webpage to contact the ITO.

Aims

- Get you extra experience with the DICE system, and its use for C programming.
- Get you to practise 'program planning' in advance of writing code. We will write a template file for programming in general.
- Get you to write code to solve a series of arithmetic tasks of increasing difficulty.
- Give you extra experience with the `if`-statement and with using different boolean conditions as the test within `if`.
- Give you more practice with `printf` and `scanf`.

You will need to complete *everything* in this Lab sheet in order to establish a firm foundation for C programming.

Prerequisites

You should have attended your previous Lab session, and completed Lab sheet 1 during that session. *If not*, please catch up on Lab sheet 1 before proceeding with this one.

Stage A Getting started

- Log into on of the DICE computers.
- Once you are logged-in, you will see your Desktop appear. The first thing to do we will do is *open a terminal window*. Do this as you did in Lab 1 – if you have forgotten how to do it, go back to Lab 1 and revise it.

  You might already have a Terminal window, restored from your previous session. If so, you can use it, but make sure you're in the right directory – see below.
- If you have created a new Terminal window, you will be in your own home directory to start.

  If you are using a Terminal window left on your Desktop from last time, this *might not* be the case. Type

  `pwd`↲

  at the Terminal prompt to see your *current directory*. If it prints something like
  `/afs/inf.ed.ac.uk/user/s`*nn*`/`*yourUUN*
  then you are at your home directory. If you *do not* see that, you are somewhere else: type

  `cd`↲

  at the terminal prompt to return to your home directory.
- From the home directory, create a subdirectory called `lab2` by entering the following from the command line:

  `mkdir lab2`↲

  Then move into that sub-directory by entering the following from the command line:

  `cd lab2`↲
- Now we are ready to start *programming*. We are going to write a few simple arithmetic programs in this lab.
  - Develop your programs carefully
  - Please ask for help when you want it!
  - Try to tick off each of the tasks.

Stage B Program Planning

In this stage we are going to develop a file called `template.c` to use as a basis for many programs. We will never actually *compile* or *run* this file – it is just a starting point to think about our programming.

- Let us first create the file `template.c`. To do this we type the following at the terminal prompt:

  `emacs template.c &`↵

  This will load up emacs 'in the background' and automatically create the (empty to start) file `template.c` (which will be stored within the `lab2` directory, since we started there).

- `main`: A key tenet of C programming is that *We always have exactly one* `main`.

  `main` always has return type `int` and usually takes no arguments (ie, input parameters are `(void)`).

  The other thing to remember is that the final thing done by `main` is to return a value, usually `EXIT_SUCCESS`. We will see some examples later in CP where we want to return `EXIT_FAILURE` sometimes, but *for now*, let us assume we always finish by returning `EXIT_SUCCESS`.

  So type the following into `template.c` as a starting 'shell' for all programs.

```
int main(void) {



  return EXIT_SUCCESS;
}
```

- *header files:* In class we have been discussing header files that will be used in C-programming, because so many basic things have been 'farmed out' to standard libraries (rather than belonging to C itself).

  We are using the constant `EXIT_SUCCESS` in our programs, hence we are making use of the `stdlib` library. Therefore we must 'include' the header file `stdlib.h` at the top of our program.

  Also almost all our programs will take input and give output in communication with the standard Input/Output stream. For example we almost always use the `printf` function, and often use `scanf`. These functions belong to the `stdio` library. Therefore we must 'include' the header file `stdio.h`.

  We use `#include` to include header files in our program, and because `stdio.h` and `stdlib.h` are standard C libraries supported by DICE, we use angled brackets to refer to them, eg `<stdio.h>`. The following `#includes` will belong to almost every program, *and always appear at the very top of your* `.c` *file, above the* `main`. Please add these two lines at the very top of `template.c` – note there should be no `;` used with `#include`.

```
#include <stdio.h>
#include <stdlib.h>
```

  There may *sometimes* be other header files to include:

  - If we are using `sqrt` or another function from the `math` library, we 'include' `<math.h>`
  - Later in the course, we will provide you with some *non-standard* `.h` and `.c` files to save to your own directory. You will have to 'include' the header files explicitly to use these files.

- Between the starting `{` and the `return EXIT_SUCCESS;` of your `main`, there is a lot of scope for creativity – this part varies widely depending on the task to hand. However, *all interesting programs have some variables*. Variables should be declared at the beginning of the `main`.

  Please mark this just by adding a comment on the line *directly below* `int main(void) {`

  `/* Variable declarations */`

- Save `template.c` and close your `emacs` window.

Stage C Converting Temperatures

The two main scales for representing weather temperatures are Celsius (used in most of the world) and Fahrenheit (used in the USA and a few small USA-dominated countries).

The Celsius scale was originally normalized with respect to the freezing and boiling points of water: the value 0 °C is the freezing point of water, and 100 °C is the boiling point of water.

The Fahrenheit scale is an alternative scale devised around the same time as Celsius – see Wikipedia for its history. It has a *linear relationship* to Celsius, expressed by the mathematical equation:

$$F = C \times (9/5) + 32.$$

Your mission, in this initial task, is to create a program which will take an integer value for Celsius as input from the user, compute the equivalent value in Fahrenheit, and print this out to the user.

Steps in programming:

- Write your code in a program called `celsius.c`.

  There is an easy way to 'start off' this file, using the Unix copy command `cp` from the Terminal prompt as follows:

  `cp template.c celsius.c`↲

  Then subsequently open `celsius.c` in emacs to develop your code further:

  `emacs celsius.c &`↲

- Decide which variables you will use first of all.

  Declare those variables *at the beginning* of the `main` function. Try to give them meaningful names.

  You will need an `int` variable for the input.

  What about the variable for the Fahrenheit value? Should we use `double` or is `float` ok in this case? If you remember the discussion from Tuesday's lecture, you will see that for this purpose, a `float` would be fine; however, as we discussed, it's easier to just use `doubles` all the time. So use a `double`.

- Use of `scanf` to enter input: remember to use the 'magic ampersand' before the variable when 'scanning in' – for example, to read an integer into the `int` variable `x`, we use

  `scanf("%d", &x);`

- 'Casting' of variables – you will be getting an `int` as input, but creating a value which is `double`. We must make sure to 'cast' the variable using `(double)`, and/or to work with 5.0, 9.0, 32.0 when

computing the equivalent in Fahrenheit (we want to make sure that / does not get interpreted as integer division)

- `printf` and dealing with 'decimal digits':

  If we have a `double` variable `y` and we output it with the C command

  ```
  printf("real number %f\n", y);
  ```

  then we will get several (six, in fact) digits after the decimal point. For some applications, we would rather not do this. We can specify 'only two decimal digits' by changing the format of our `printf` as follows:

  ```
  printf("real number %.2f\n", y);
  ```

- Now develop your program.

  The slides from the week 2 lectures will also be helpful.

- Save your program when you are happy with it. Compile your program from the terminal prompt:

  `gcc -Wall celsius.c`↲

  There may be some *debugging* to be done.

  Please read the output error/warning messages very carefully, to debug your program. If there are no messages, your program compiled successfully (but could still have bugs!).

- When compilation is successful, you will have a `a.out` file in your directory.

  Type `./a.out`↲ to *run* your program and test its results!!! You may need to modify the program further.

- Next, move your 'executable' for `celsius.c` into a different file, for example, call it `celsius` (this is different to `celsius.c`) by using the Unix 'move' command `mv` from within your `lab2` directory:

  `mv a.out celsius`↲

  This will rename `a.out` to be `celsius`.

  We do this because when we write our other programs inside the directory `lab2` and compile them, the 'executable' will get saved to `a.out` as always. So once you are happy with the executable for `celsius.c`, make a copy. Then in the future, to run the celsius program, you will type the following:

  `./celsius`↲

- Finally, submit your program to us by

  `submit cp lab2 celsius.c`↲

  Note, by the way, that you can re-submit your lab programs as often as you like – we only keep the last one.

Your result for `celsius.c`

- ☐ A program which requests input with `printf`, reads the input with `scanf`, and outputs with `printf`
- ☐ Program should be *correct*. To test:

  Given the value `0` for Celsius, should print `32.00`

  Given the value `32` for Celsius, should print `89.60`
- ☐ Can you make your program give output with three decimal digits? With no decimal digits?
- ☐ Make sure you move `a.out` into `celsius` before proceeding with the other tasks.
- ☐ Remember to submit your program.

Stage D Converting temperatures II

Next write a program which takes as input an integer representing some Fahrenheit temperature, and prints the corresponding Celsius value to *two decimal digits*. Write your program in a file called `fahrenheit.c` (again, you will probably want to start by copying over `template.c` – or if you're lazy, by copying `celsius.c`!). There is *an extra initial problem-solving stage* to this task. You need to work out the conversion equation with F on the left-hand side first.

Your result for `fahrenheit.c`

- ☐ Program should be *correct*. To test:

  Given the value 32 in Fahrenheit, should return 0.00

  Given the value 0 on Fahrenheit, should return -17.78
- ☐ Make sure you rename `a.out` into a file `fahrenheit` (or similar) before proceeding with other programs.
- ☐ Submit your program with

  ```
  submit cp lab2 fahrenheit.c↲
  ```

Stage E What day was it?

Note: *This is quite a challenging task at this stage in the course. Don't worry if you don't complete it in this lab session. If you can, finish it in your own time, or at the start of the next lab.*

The goal is for you to write a program called `whatday.c` which takes a date in 2017, and tells you the day it corresponds to. You will use the following facts in solving this task:

- The input is to be given as two integers, the day of the month, and then the month in integer format (i.e., a number from 1 to 12).
- The 1st of January was a Sunday in 2017.
- In 2017, most months have 31 days. February (month 2) has 28 days. April (4), June (6), September (9) and November (11) have 30 days each.

There are two stages to this task:

- The problem-solving aspect of this task, where we work out the steps we need in order to get the day. At a minimum we will need to do the following things:

  (i) Work out *how many days* have passed since the 1st of January, and the date of interest. That will mean adding up the days for any months which have already gone past, plus any extra days from the current month.

  For example, if the date of interest was 5th March, we would add 31 for January, 28 for February, and 4 days for the difference between 5th and 1st.

  A slightly long-winded but easy to understand way is to write a series of `if`-statements, each adding the number of days for that month (if the month-value-read-in is greater than the current month for that `if`).

  If you find this easy, try thinking about short ways to write it – but only if you find it easy!

  (ii) Once you have the total number of days, you will use the `%` 'mod' operator (mentioned in slides for Lecture 4) with respect to 7 to compute the offset (in terms of days of the week) of your date-of-interest from 1st January (which was a Sunday).

  (iii) Finally, you will need a long `if...else` statement in our 'general form' to output the correct day, depending on the offset from Sunday. (Again, there are easy ways to do this, but we haven't talked about them yet – feel free to explore.)

- It will be helpful to write out the details of the problem-solving process by hand (even in English). But afterwards, the second stage is to write this procedure (or *algorithm*) as C code.

  Try to write the C for each stage, one stage at a time.

  A couple of issues will be:

  – Remember as always you can create a 'shell' for `whatday.c` by copying over `template.c` in the usual way.

  – Your variables, including variables for the inputs of day and month, will need to be declared below the `main`.

  To do (i), you will certainly need to define an extra `int` variable to store the count of days. You may need extra variables, depending how exactly you are solving the task.

  – In computing the *number of days since 1st January* in (i), you will need to use the `if...else` statement *somehow*, because different months do not all have the same number of days.

It will *help* shorten your code if you use more complex *Boolean conditions* using `&&` or `||` within the `if` statements, as described in lecture slides 5 (these slides are online in advance).

- Will you need to include to use any functions from the `math` library (and hence 'include' `math.h`) or are the core arithmetic operations enough for this task?
- The usual commands are used to do the compilation and execution of the program. To compile:

  ```
  gcc -Wall whatday.c↵
  ```

  Once you have compiled your program (and the executable has been created in `a.out`) you can run it as usual:

  ```
  ./a.out↵
  ```

  Once you have debugged your program against the tests below, save it for future execution in a file called `whatday` (different from `whatday.c` because the `.c` is missing) as follows:

  ```
  mv a.out whatday↵
  ```

- Submit your program using

  ```
  submit cp lab2 whatday.c↵
  ```

If you still have time left, you might want to consider extending your program, to check that the inputs were legal (ie the months is between 1 and 12, the day is greater than 1 and less than the maximum for the month), so that you can output an error message for illegal dates.

This would involve further use of the `if` statement.

'What day' program

☐ Testing.

You should get the answer

It was a Monday

in response to inputting 5 for day, and 6 (June) for month.

You should get the result

It was a Tuesday

in response to inputting 8 for the day, and 8 (August) for the month.

You should get the result

It was a Wednesday

in response to inputting 4 for the day, and 1 (January) for the month.

Do further tests to check correctness.

☐ Submit your working program before ticking this box.

Julian Bradfield after *Mary Cryan, 28 Sept 2016*