

Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Winter 2015

Lab 3 - C Program Translation, Modules, and Memory Diagrams

Objectives

- To understand the preprocessing, compilation and linkage steps that occur when the source code in a C program is translated to an executable program;
- To understand how C modules are implemented using .c and .h files;
- To learn how to draw diagrams that represent the state of an executing C program.

Attendance/Demo

To receive credit for this lab, you must make reasonable progress towards completing the exercises and demonstrate the code you complete. **Also, you must submit your work for Exercise 3 to cuLearn by the end of the lab period.** (Instructions are provided in the *Wrap Up* section at the end of this handout.)

When you have finished all the exercises, call a TA, who will review the code you wrote. For those who don't finish early, a TA will ask you to demonstrate whatever code you've completed, starting about 30 minutes before the end of the lab period. **Any unfinished exercises should be treated as "homework"; complete these on your own time, before your next lab.**

Part 1 - Program Translation

Until now, we've translated our C source code into executable programs by selecting the **Build** command in Pelles C. Although this is convenient, it hides the separate steps that take place during this translation process. You'll learn about these steps in Exercises 1 and 2.

When a .c file is compiled:

- The C *preprocessor* processes all *directives* (statements that begin with #, such as `#include`);
- The *compiler* then checks the syntax of the C statements. If there are no syntax errors, the compiler translates the C statements into the corresponding CPU instructions. The output of this step is known as *object code* and is stored in an *object file*.

These two steps are repeated for every .c file in a project. For every .c file, a separate object file is produced.

Next, the *linkage* step occurs:

- A *linker* program combines the object files generated from your .c files and any required object code from the standard C library into an executable program. For example, if your code calls `printf`, the object code for this function (which is provided with the C programming tools) must be linked to your object files.

In Exercises 1 and 2, you'll investigate these translation steps in more detail, by preprocessing, compiling and linking a modular C program.

The program used in these exercises deals with this problem:

Consider an object moving with an initial velocity u that is subject to a constant acceleration a which is aligned in the same direction as the initial velocity. The object's final velocity v after an elapsed time t is given by the formula $v = u + at$.

Exercise 1

1. Create a new folder named Lab 3.
2. Download `motion_ex1.zip` from cuLearn to your Lab 3 folder. Right-click on the icon for this file, and when a pop-up menu appears, select **Extract All...** This will create a folder named `motion_ex1`. Look in this folder. It should contain a Pelles C Project File, `motion_ex1.ppj`, plus three C source files: `main.c`, `motion.c` and `motion.h`.
3. This project has been configured so that, when the .c files are compiled individually, compilation stops after the preprocessing phase. **Important: while working on this exercise, do not select the Build or Rebuild menu commands from the Project menu, or click the Build button.**
4. Double-click `motion_ex1.ppj` to launch Pelles C and open the project. (Make sure you use the uncompressed folder, `motion_ex1`. Don't work out of the compressed folder; i.e., the zip file.)
5. In the project window (on the left or right-hand side of the IDE), locate the icon for `motion.h` in the Include files folder. Double-click this icon to open `motion.h` in an editor window. Read the code - you'll see that this file declares the prototype for a function named `calculate_velocity`.
6. Find the icon for `motion.c` in the project window. Double-click this icon to open `motion.c` in an editor window. Read the source code for this module. You'll see the preprocessor directive:

```
#include "motion.h"
```

followed by the complete definition of the `calculate_velocity` function.

7. From the menu bar, select **Project > Compile motion.c**. (Again, do not select the

Build or Rebuild commands.) The C preprocessor will be run on this file, then compilation will stop. In the project view, you will see a message stating that POCC (the C compiler) has compiled `motion.c`.

8. From the menu bar, select **File > Open...** and navigate to the `motion_ex1` folder. From the drop-down menu, pick **All files (*.*)**. Open `motion.i`. This file contains the output from the C preprocessor when it processed `motion.c`. Read the source code. Notice that the `#include` statement in `motion.c` has been replaced with the contents of `motion.h`.
9. Find the icon for `main.c` in the project window. Double-click this icon to open `main.c` in an editor window. Read the source code - you'll see the preprocessor directives:

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

and

```
#include "motion.h"
```

followed by the definition of a `main` function that calls `calculate_velocity`.

10. From the menu bar, select **Project > Compile main.c.** (**Once again, do not select the Build or Rebuild commands.**) The C preprocessor will be run on this file, then compilation will stop. In the project view, you will see a message stating that POCC (the C compiler) has compiled `main.c`.
11. From the menu bar, select **File > Open...** From the drop-down menu, pick **All files (*.*)**. Open `main.i`. This file contains the output from the C preprocessor when it processed `main.c`. Read the source code. Notice that the `#include <stdio.h>` and `#include "motion.h"` statements in `main.c` have been replaced with the contents of `stdio.h` and `motion.h`, respectively. If you scroll down far enough, you'll eventually see the definition of `main`.
12. Close this project.

Exercise 2

1. Download `motion_ex2.zip` to your Lab 3 folder. Right-click on the icon for this file, and when a pop-up menu appears, select **Extract All...** This will create a folder named `motion_ex2`. Look in this folder. You should have a Pelles C Project File, `motion_ex2.ppj`, plus three C source files: `main.c`, `motion.c` and `motion.h`. (These are the same files that you used in Exercise 1.)
2. This project has been configured so that debugging information is generated during the compilation and linking phases. **Important: while working on this exercise, do not select the Build or Rebuild menu commands from the Project menu, or click the Build button, unless you are instructed to do so.**

3. Double-click `motion_ex2.ppj` to launch Pelles C and open the project. (Make sure you use the uncompressed folder, `motion_ex2`. Don't work out of the compressed folder; i.e., the zip file.)
4. Find the icon for `motion.c` in the project window. Double-click this icon to open `motion.c` in an editor window.
5. From the menu bar, select **Project > Compile motion.c**. The C preprocessor will generate `motion.i` from `motion.c`. Next, the C compiler will compile `motion.i` and generate an object code file. In the project view, you will see a message stating that POCC (the C compiler) has compiled `motion.c`.
6. Look in the `motion_ex2` folder. It will now contain a folder named `output`. Look inside this folder. It now contains a file named `motion.obj`. This is the *object file* containing the compiled C code; that is, the CPU instructions that correspond to the C statements in `motion.c`.
7. Find the icon for `main.c` in the project window. Double-click this icon to open `main.c` in an editor window.
8. From the menu bar, select **Project > Compile main.c**. The C preprocessor will generate `main.i` from `main.c`. Next, the C compiler will compile `main.i` and generate an object code file. In the project view, you will see a message stating that POCC (the C compiler) has compiled `main.c`.
9. Look in the `output` folder. It now contains a file named `main.obj`. This is the *object file* containing the CPU instructions that correspond to the C statements in `main.c`.
10. At this point, Pelles C has compiled the two C source files into two object files, but it has not yet linked the object files into an executable program. There doesn't appear to be a way to run the linking phase as a separate step from within the Pelles C IDE. Instead, we'll rebuild the entire project; i.e., recompile both C files, then link them into an executable program. From the menu bar, select **Project > Rebuild motion_ex2.exe**. Do not select the **Build** menu command, or click the **Build** button.
11. In the project view, you will see a message stating that POCC (the C compiler) was run twice, because `main.c` and `motion.c` were compiled independently. This will be followed by a message stating that POLINK (the linker) was run. The linker produces an executable program, by linking `main.obj` and `motion.obj` to object files in the standard library (e.g, the object code for `printf`).
12. Look in the `motion_ex2` folder. It will now contain a file named `motion_ex2.exe`. This is your executable program.
13. From the menu bar, select **Project > Debug motion_ex2.exe**. A window labelled **Debugger - Stopped** will open, displaying the source code from `main.c`.
14. Move the mouse cursor anywhere in this window, then right-click. From the pop-up

menu, select **Show disassembly**. Right-click again, and select **Show code bytes**. You'll see the sequence of CPU instructions that correspond to each C statement, represented using *assembly language* mnemonics. To the left, you'll see the hexadecimal (base-16) numbers that encode these CPU instructions. When you run a program, these numbers are fetched by from memory by the CPU, decoded as instructions, and executed. (You'll learn more about CPU instruction encoding and execution in SYSC 2001 or SYSC 3006, depending on your degree program.)

15. From the menu bar, select **Debug > Go** to execute the program.

16. Close this project.

Part 2 - Implementing Modules in C

Every C module consists of two files: a header (.h) file and a .c file. A header file usually contains *function prototypes*, but not the complete definitions of functions. The header file must be included (via the `#include` directive) in the .c file that contains the function *definitions*, and in any .c file that *calls* the functions.

For example, the project in Part 1 contains a module named `motion`. The header file, `motion.h`, contains the *function prototype* for `calculate_velocity`:

```
double calculate_velocity(double initial_v, double accel,
                        double time);
```

Function `calculate_velocity` is defined in `motion.c`:

```
#include "motion.h"

double calculate_velocity(double initial_v, double accel,
                        double time)
{
    return initial_v + accel * time;
}
```

Notice that `motion.c` contains the preprocessor directive, `#include "motion.h"`. As you saw in Part 1, during the preprocessing phase, this directive is replaced with the contents of `motion.h`. This means that the output of the preprocessor, `motion.i`, will contain both the prototype and the definition of `calculate_velocity`.

When `motion.i` is compiled, the compiler will verify that the declaration of `calculate_velocity` is consistent with the function's definition; i.e., the return type and parameter list in the function prototype match the return type and prototype in the function definition.

Function `main` in file `main.c` calls `calculate_velocity`:

```
calculate_velocity(initial_velocity, acceleration, time)
```

When `main.c` is compiled, the C compiler must verify that the function call is correct; however, the function isn't defined in `main.c`. That's why `main.c` has the preprocessor directive `#include "motion.h"`. When `main.c` is compiled, the compiler will then verify that the call to `calculate_velocity` is consistent with the declaration of `calculate_velocity` (as specified by the function prototype in `motion.h`).

To summarize: placing the prototype for `calculate_velocity` in `motion.h`, and `#include`'ing this file in the `main.c` and `motion.c`, allows Pelles C to verify that the function declaration (prototype), the function definition and the call to the function, are all consistent.

In Exercise 3, you'll add a function to the `motion` module, and modify `main` to call this function.

Exercise 3

Consider an object moving with an initial velocity u that is subject to a constant acceleration a which is aligned in the same direction as the initial velocity. The object's displacement s after an elapsed time t is given by the formula:

$$s = ut + \frac{1}{2}at^2$$

You'll now going to add a function named `calculate_displacement` to the `motion` module. This function will have three parameters: the object's initial velocity (measured in m/s), its constant acceleration (measured in m/s²), and an elapsed time (measured in s). This function will return the object's displacement after the elapsed time.

1. Create a new project named `motion_ex3` inside the `Lab 3` folder. The project type must be `Win32 Console program (EXE)`. You should now have a folder named `motion_ex3` inside your `Lab 3` folder. Check this. If you do not have a project folder named `motion_ex3`, close this project and repeat Step 1.
2. Copy/paste `main.c`, `motion.c` and `motion.h` from your `motion_ex2` folder to your `motion_ex3` folder. **You must also add `main.c` and `motion.c` to your project:** from the menu bar, select `Project > Add files to project...` In the dialogue box, select `main.c`, then click `Open`. An icon labelled `main.c` will appear in the Pelles C project window. Now add `motion.c` to the project. Pelles C will automatically add `motion.h` to the project.
3. Build the project. It should build without any compilation or linking errors.
4. Edit `motion.h`, so that it contains the declaration of the function prototype for `calculate_displacement`. You have to decide what the function's return type should be, as well as the names and types of the three parameters. Use descriptive names that will help someone else understand your code - don't name the parameters `u`, `a` and `t`. (Don't delete the prototype for `calculate_velocity`.)

5. In `motion.c`, define the complete implementation of the `calculate_displacement` function. (Don't delete the definition of `calculate_velocity`.)
6. Add code to `main` to exercise `calculate_displacement` (don't delete the code that exercises `calculate_velocity`). Your code should print the values of the function's three arguments, the expected result, and the actual result returned by the function. (Read the code in `main` that exercises `calculate_velocity` for an example of how to do this.)
7. Compile and execute your modified version of the program. Verify that the value returned by `calculate_displacement` is correct.
8. You'll need to submit a compressed folder containing the project for Exercise 3. See the *Wrap Up* section at the end of this handout.

Part 3 starts on the next page.

Part 3 - Tracing Code/Memory Diagrams

Fibonacci numbers are defined by the following formulas:

$$F_1 = 1$$

$$F_2 = 1$$

$$F_n = F_{n-1} + F_{n-2}, n > 2$$

Also, it is conventional to define F_0 as 0. The Fibonacci sequence for $n = 0, 1, 2, 3, 4, 5, 6, 7, 8 \dots$ is therefore 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Here is a definition of a C function that is passed n and returns F_n , for $n \geq 0$.

```
int fibonacci(int n)
{
    if (n == 0)    // fib(0)
        return 0;

    if (n == 1)    // fib(1)
        return 1;

    int temp1 = 0;
    int temp2 = 1;
    int nextfib;

    for (n = n - 2; n >= 0; n = n - 1) {
        nextfib = temp1 + temp2;
        temp1 = temp2;    /* Point A. */
        temp2 = nextfib;
    }
    return nextfib;    /* Point B. */
}
```

Here is the definition of a main function that calls fibonacci:

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

int main(void)
{
    int result;
    result = fibonacci(5);
    printf("fib(5) = %d\n", 5, result);    /* Point C. */
    exit(0);
}
```


The TAs will provide you with a worksheet. On this worksheet, using the notation presented in lectures, draw three separate memory diagrams on the worksheet, one each for parts (a), (b) and (c). *Do not combine your solutions into a single diagram.* Do not use arrows to depict the flow of data into and out of variables. Remember, arrows are only used to depict pointers.

- (a) Draw a memory diagram that depicts the program's activation frame(s) immediately **after** the statement at Point A is executed for **the first time**; that is, immediately after

```
temp1 = temp2;
```

is executed during the first iteration of the `for` loop.

- (b) Draw a memory diagram that depicts the program's activation frame(s) after control has left the `for` loop, but immediately **before** the statement at Point B is executed; that is, just before the `return` statement is executed.

- (c) Draw a memory diagram that depicts the program's activation frame(s) immediately **before** the statement at Point C is executed; that is, just before the `printf` call is executed.

To double-check your work, copy the program into a new Pelles C project, then use the debugger to single-step through the program, observing the function parameters and local variables as each statement is executed. (Instructions on using the debugger were presented in Lab 2.)

Wrap-up

1. Show the TA the memory diagrams you prepared for Part 3. Do not hand in this worksheet for grading. Solutions will be posted after all lab sections have completed this lab. Remember to have a TA review your solution to Exercise 3, assign a grade (Satisfactory, Marginal or Unsatisfactory) and have you initial the demo/sign-out sheet.
2. The next thing you'll do is package the project from Exercise 3 in a ZIP file (compressed folder) named `motion_ex3.zip`. To do this:
 - 2.1. From the menu bar, select **Project > ZIP Files...** A **Save As** dialog box will appear. If you named your Pelles C project `motion_ex3`, the zip file will have this name by default; otherwise, you'll have to edit the **File name:** field and rename the file to `motion_ex3` before you save it. **Do not use any other name for your zip file** (e.g., `lab3.zip`, `my_project.zip`, etc.).
 - 2.2. Click **Save**. Pelles C will create a compressed (zipped) folder, which will contain copies of the the source code and several other files associated with the project. (The original files will not be removed). The compressed folder will be stored in your project folder (i.e., folder `motion_ex3`).
3. Before you leave the lab, log in to cuLearn and submit `motion_ex3.zip`. To do this:

- 3.1. Click the **Submit Lab 3** link. A page containing instructions and your submission status will be displayed. After you've read the instructions, click the **Add submission** button. A page containing a **File submissions** box will appear. Drag **motion_ex3.zip** to the **File submissions** box. **Do not submit another type of file (e.g., a Pelles C .ppj file, a RAR file, a .txt file, etc.)**
- 3.2. After the icon for the file appears in the box, click the **Save changes** button. At this point, the submission status of your file is "**Draft (not submitted)**". If you're ready to finish submitting the file, jump to Step 3.4. If you instead want to replace or delete your "draft" file submission, follow the instructions in Step 3.3.
- 3.3. You can replace or delete the file by clicking the **Edit my submission** button. The page containing the **File submissions** box will appear.
 - 3.3.1. To overwrite a file you previously submitted with a file having the same name, drag another copy of the file to the **File submissions** box, then click the **Overwrite** button when you are told the file exists ("**There is already a file called...**"). After the icon for the file reappears in the box, click the **Save changes** button.
 - 3.3.2. To delete a file you previously submitted, click its icon. A dialogue box will appear. Click the **Delete** button., then click the **OK** button when you are asked, "**Are you sure you want to delete this file?**" After the icon for the file disappears, click the **Save changes** button.
- 3.4. Once you're sure that you don't want to make any changes, click the **Submit assignment** button. A **Submit assignment** page will be displayed containing the message, "**Are you sure you want to submit your work for grading? You will not be able to make any more changes.**" Click the **Continue** button to confirm that you are ready to submit your lab work. This will change the submission status to "**Submitted for grading**".