

# ENCM 369 Winter 2018 Lab 1 for the Week of January 15

Steve Norman  
Department of Electrical & Computer Engineering  
University of Calgary

January 2018

Lab instructions and other documents for ENCM 369 can be found at  
<http://people.ucalgary.ca/~norman/encm369winter2018/>

## Administrative details

### Each student must hand in his or her own assignment

Later in the course, you will be allowed to work in pairs on some assignments.

### Due Dates

The Due Date for this assignment is 3:30pm Friday, January 19.

The Late Due Date is 3:30pm Monday, January 22.

The penalty for handing in an assignment after the Due Date but before the Late Due Date is 3 marks. In other words,  $X/Y$  becomes  $(X-3)/Y$  if the assignment is late. There will be no credit for assignments turned in after the Late Due Date; they will be returned unmarked.

### Marking scheme

B	6 marks
G	6 marks
H	4 marks
total	16 marks

### How to package and hand in your assignments

Please make sure all your work is easy to read and conveniently organized; this will be a *big help* to the teaching assistants who have the difficult job of marking your assignments.

Make a cover page with this information on it, in large, easy-to-read handwriting or printing, ...

- your *name* (but not your ID number)
- your *lab section number* (B01, B02, B03 or B04)
- the name of this course—*ENCM 369*
- which lab assignment it is—*Lab 1* this week, *Lab 2* next week, and so on

*Staple* your assignment together. Assignments held together by paper clips or by folding corners of pages together will almost certainly fall apart in the collection boxes, and teaching assistants are not responsible for re-collating loose pages they might find.

Please check carefully that you are handing in *all* of the material you are supposed to be handing in. “I forgot” is not an acceptable excuse for handing in part or all of an exercise after the Due Date.

Hand in your assignment in the box for your lab section in the west hallway on the 2nd floor of the ICT Building. *Before letting your assignment fall in to the box, please double-check that you are handing it in to the correct box!*

## Editing and running C programs using a text editor and the gcc command line

In many lab assignments in ENCM 369, you will be expected to work with C code in some way, so it is important that you know how to edit C code, and build executables from it in the lab.

Most students in ENCM 369 in Winter 2018 took ENCM 339 in Fall 2017 or Spring 2017; those students should be quite familiar with using text editors such as Geany or Notepad++ to edit C source code, and using Cygwin Terminal to build and run executables. However, there may be a few students in ENCM 369 this term who have transfer credit for ENCM 339 and may not be familiar with the tools used for programming in that course.

Instead of inserting into this document a lot of text about how to edit and run C programs in the lab, I'll just post a link to the “Lab Information” page for Section 01 of ENCM 339 in Fall 2017:

[http://people.ucalgary.ca/~norman/encm339fall2017/lab\\_information/](http://people.ucalgary.ca/~norman/encm339fall2017/lab_information/)

You should consider clicking on that link and doing some reading and coding if one or the other of the following is true:

- you're not at all familiar with the programming tools used in recent versions of ENCM 339;
- you took ENCM 339 in Spring or Fall 2017, but you did most of your lab work on your own computer, and you would like to review the basics of editing and running C code in ICT 320.

The key documents to read are “Editing and Running Programs in the ENCM 339 Lab” and the Lab 1 Instructions.

## Exercise A: Practice editing C code and building an executable

### Read This First

There are no marks for this exercise, but I *strongly* recommend that you make a serious effort to get it done before your first lab period. If you are unsuccessful, please be ready to ask questions at the *beginning* of the lab period.

### What to do, Part I

If necessary, read the instructions for ENCM 339 Fall 2017 Section 01 Lab 1, and work through some of the exercises.

## What to do, Part II

Open a Cygwin Terminal window. Within `/cygdrive/h`, create a directory called `encm369`. Within your `encm369` directory, create a directory called `lab01`, and within that create a directory called `exA`.

Figure 1: C code for Exercise A.

```
#include <stdio.h>

int main(void)
{
    int n = 0;
    int power = 1;

    printf("%12s%12s%12s\n", "n", "2**n", "2**n");
    printf("%12s%12s%12s\n", "decimal", "decimal", "hex ");
    while (n <= 16) {
        printf("%12d%12d%12x\n", n, power, power);
        power = power + power;
        n++;
    }
    return 0;
}
```

Start up Geany or Notepad++ (or some other text editor, if you know that text editor well). Type in the first line from Figure 1, then use a **Save as ...** command to save the file with name `exA.c`, in your `encm369/lab01/exA` directory. Use the `ls` command to confirm that a file called `exA.c` exists within that directory.

Return to the text editor, and finish typing in the rest of the code from Figure 1. When you have finished, save the file and return to your terminal window.

In the terminal window, build an executable with the command

```
gcc -Wall exA.c
```

If there are errors or warnings, go back to the text editor, try to fix the errors, then try again to build an executable. If there are no errors or warnings, run the command

```
./a.exe
```

You should see a table of powers of  $2^n$  for values of  $n$  from 0 to 16, displayed in both base ten and base sixteen.

*Keep working on this exercise until you are sure you know what you are doing—if you do not, you will have serious problems with all of the remaining exercises!*

## What to Hand In

Nothing.

## How to download files for ENCM 369 labs

Most lab exercises in this course will require you to download one or more files containing C code, assembly language code, or data.

Links for downloading these files can be found on the same Web page where you find lab instructions. You have the option of clicking through links to get files one at a time or downloading all the files you need for one lab in a single `.zip` archive.

When you are told to make a copy of, for example,

```
encm369w18lab01/exB/globals.c,
```

you should look for it starting on the ENCM 369 Lab Information web page.

## Exercise B: Global variables, review of pointer arithmetic

### Read This First

Most variables you saw or created in programs in ENCM 339 were *local variables*. A local variable can be directly accessed only within the function definition in which it appears. (A local variable of one function can be *indirectly* accessed by another function, if that other function has a pointer to the variable.)

C variables declared outside of all function definitions in a C source file are called *global variables* or *external variables*. Unlike local variables, global variables can be directly accessed from many different function definitions.

There are two kinds of global variables: those declared to be `static` and those *not* declared to be `static`. Global variables declared to be `static` can be directly accessed by all functions in the source file where the variable is declared. (This use of the keyword `static` is confusing, because it has nothing to do with static memory allocation.) Global variables *not* declared to be `static` can be directly accessed by all functions in a program.

(If you are working on a program where a global variable is in fact accessed from more than one source file, you need to know the rules regarding the `extern` keyword. See a good C text for details, and read carefully.)

Global variables, regardless of whether they are declared as `static`, are always statically allocated. So, according to rules you should have learned in ENCM 339, global variables are found in the static storage region, are allocated and initialized before `main` starts, and remain allocated as long as the program runs.

Unlike *automatic* variables (which are the “normal” kind of local variables of functions) global variables that don’t have initializers are initialized to zero. For example, consider this variable declaration:

```
int arr[5];
```

If `arr` were local to some function, you could not count on the elements of `arr` to have any particular values. But if `arr` were a global variable, its elements would all be initialized to zero.

### What to do

#### Part 1

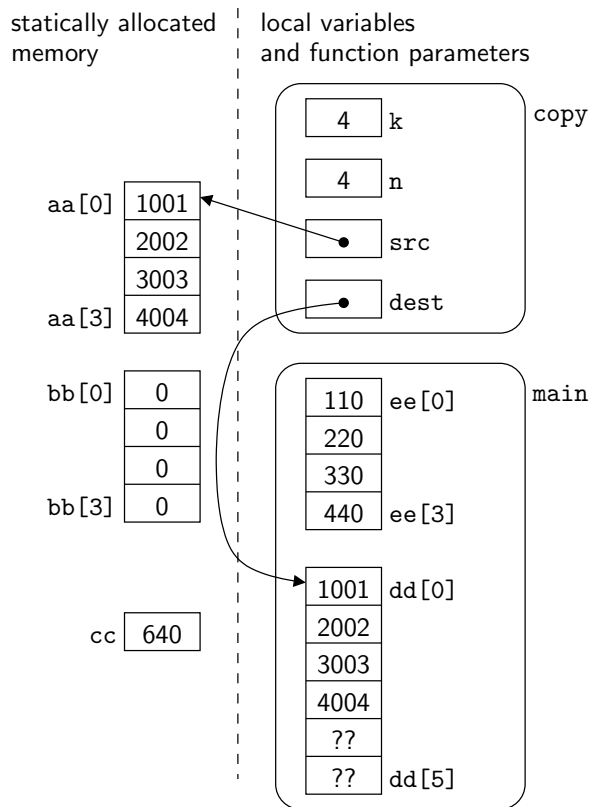
Copy the file `encm369w18lab01/exB/globals.c`

Study the file carefully and make sure you understand the diagram of program state at **point one** given in Figure 2.

Make a similar diagram to show the state of the program when it reaches **point three**. (*Note:* In ENCM 369, whenever you are asked to hand in diagrams, you may draw them neatly by hand or by computer, whichever you find more convenient.)

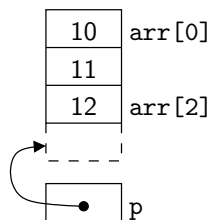
Figure 3 should give you a hint about how to draw part of the diagram.

**Figure 2:** The state of the Exercise B program at the moment when `point one` is reached. At that moment, functions `update_cc` and `reverse` are not active, so their parameters and local variables do not exist. The notation `??` is used to indicate uninitialized local variables. **Important note for ENCM 369:** As the course progresses, we will see that some of the local variables and parameters of a program are in a region of main memory called *the stack*, and other local variables and parameters are in *registers*; we'll see later in the course that for MIPS the arrays `dd` and `ee` would be allocated on the stack, but the integers and pointers belonging to `copy` would be in general-purpose registers, *not* in memory at all!



**Figure 3:** Example of a pointer pointing “just beyond” the last element of an array. Here the *element* `arr[3]` does not exist, but the *address* of `arr[3]` *does* exist. This concept is useful for setting up a condition to quit a loop, when the loop uses pointer arithmetic to step through an array.

```
// fragment of a C function
int arr[] = { 10, 11, 12 };
int *p;
p = arr + 3;
```



**Figure 4:** Table for Exercise B, Part 2.

instant in time	to	from	pastend
point two, first time	0x100090		
point two, second time			
point two, third time			
point two, last time			

**Part 2**

Instead of using arrows to show what addresses are contained by pointers, let's track pointers as actual numbers.

We'll assume that the size of an `int` is 4 bytes. We'll use hexadecimal numbers, writing them in C language notation. For example, `0x7ffe18` represents a number we would have written as  $7FFE18_{16}$  in ENEL 353.

Suppose that the address of `ee[0]` is `0x7ffe78`. Then the addresses of elements 1, 2, 3 in the array must be `0x7ffe7c`, `0x7ffe80`, and `0x7ffe84`—each address is 4 greater than the previous one. (Here is a fact you will use often in this course: In base sixteen, adding `0xc` and 4 generates a sum digit of zero and a carry out of one.)

Now suppose that `&bb[0]` is `0x100090` and `&ee[0]` is `0x7ffe78`. Use that information to fill in all of the addresses in all of the cells in a copy of the table of Figure 4.

**What to Hand In**

Hand in a diagram for Part 1 and a completed table for Part 2.

**Using your own computer for Exercises C–H**

We (your instructors and your TAs) expect that most students will do Exercises C to H on Linux workstations in ICT 320. However, all of the exercises can be done on your own computer, if you have a reasonably up-to-date C development system installed.

All students should do at least two or three of the exercises in ICT 320, just to get comfortable with the process of editing C code and building executables using a text editor and a command line.

**Exercise C: goto statements****Read This First**

`goto` statements are available in C and C++, and their equivalents are available in most programming languages.

You can have a long career writing high-quality C and C++ code for a huge variety of applications and never once use a `goto` statement. Just about everything you can do with `goto` can be done more clearly some other way. Use of `goto` tends to make code unreasonably hard to understand and debug.

Nevertheless, it's useful to know what `goto` does, for the following reasons:

- (1) You might find `goto` in code that somebody else wrote.
- (2) C code generated by a program (as opposed to code written by a human being) may have `goto` statements in it.

(3) Writing `goto` statements is similar to writing branch instructions in assembly language.

It's reason (3) that makes the `goto` statement relevant to ENCM 369.

Consider the following simple program:

```
#include <stdio.h>
int main(void)
{
    int i;
    for (i = 0; i < 4; i++)
        printf("i is %d.  ", i);
    printf("\n");
    return 0;
}
```

Obviously, the output is

```
i is 0.  i is 1.  i is 2.  i is 3.
```

Here is an equivalent program written using `goto` statements:

```
#include <stdio.h>
int main(void)
{
    int i;
    i = 0;
loop_beginning:
    if (!(i < 4)) goto past_loop_end;
    printf("i is %d.  ", i);
    i++;
    goto loop_beginning;
past_loop_end:
    printf("\n");
    return 0;
}
```

The output is exactly the same as that of the earlier program with the `for` loop.

The identifiers `loop_beginning` and `past_loop_end` are examples of what are called *labels*. A label is used to name a particular statement; a colon must appear between a label and the statement it names. A `goto` statement has the following syntax:

```
goto label;
```

A `goto` statement causes the flow of statement execution to jump to whatever statement is named by the label. This should be reasonably clear from the example.

## What to Do

Determine the output of the following program by tracing its execution line by line with a pencil and paper.

```
#include <stdio.h>
int main(void)
{
    int outer, inner;
    outer = 3;
```

```

outer_loop:
    if (outer == 0) goto quit_outer_loop;
    inner = 0;
inner_loop:
    if (inner > outer) goto quit_inner_loop;
    printf(" %d", 100 * outer + inner);
    inner++;
    goto inner_loop;
quit_inner_loop:
    printf(" **\n");
    outer--;
    goto outer_loop;
quit_outer_loop:
    printf("*****\n");
    return 0;
}

```

Check your work by typing in the program, compiling it, and running it. (If it doesn't compile, goes into an infinite loop when it runs, or crashes, you have made a typing error.)

## What to Hand In

Nothing.

## Goto-C: Read this before starting Exercises D–H

From now on in this course, the term “Goto-C” refers to a programming language that is C with the following modifications:

- The only kind of `if` statement allowed in Goto-C is the following:

```
if (expression) goto label;
```

- Goto-C does not have the following keywords and operators:

```
else while for do switch && ||
```

- Goto-C does not have `?:`, the conditional operator.

As you will soon see, Goto-C is an annoying language, significantly harder to read and write than normal C. However, working with Goto-C will help you learn how to code algorithms in assembly language.

Consider the following simple fragment of normal C:

```

if (x >= 0)
    y = x;
else
    y = -x;

```

The above is illegal in Goto-C for two reasons: the `if` statement is not in an acceptable form, and the banned keyword `else` is used. One way to translate the fragment to Goto-C is as follows:

```

if (x < 0) goto else_code;
y = x;
goto end_if;
else_code:

```



```
y = -x;
end_if:
```

**Technical Note:** The C standard does not permit code like the following:

```
void f(int i)
{
    if (condition) goto L;
    code
L:
}
```

The reason is that the label `L` is followed by a closing brace, which is not a statement. To write standard-compliant code, use an empty statement—a semicolon all by itself ...

```
void f(int i)
{
    if (condition) goto L;
    code
L:
    ;
}
```

## Exercise D: Translating `if` / `else-if` / `else` code to Goto-C

### What to Do

Copy the file `encm369w18lab01/exD/temperature.c`

Read the file `temperature.c`. Make an executable and run it. Then recode the function `report` in Goto-C. Make sure that the output of your modified program exactly matches the output of the original program.

### What to Hand In

Nothing.

## Exercise E: Simple loops in Goto-C

### Read This First

Simple C `while` and `for` loops are easy to code in Goto-C. You need a statement of the form

```
if (condition) goto label;
```

at the top of the loop to skip past the end of the loop when appropriate and you need a statement of the form

```
goto label;
```

to jump backwards after the loop body has been executed. For example, this code:

```
while (i >= 0) {
    printf("%d\n", i);
    i--;
}
```

can be written as:

```
loop_top:
    if (i < 0) goto past_end;
    printf("%d\n", i);
    i--;
    goto loop_top;
past_end:
```

Remember that the condition to quit looping is the opposite of the condition to continue looping.

## What to Do

Copy the file `encm369w18lab01/exE/simple_loops.c`

Read the file and follow the instructions in the comment at the top of the file.

## What to Hand In

Nothing.

## Exercise F: `&&` and `||` in Goto-C

### Read This First

You should recall from ENCM 339 that *short-circuit evaluation* is used when the `&&` and `||` operators are encountered in C code. (This also happens with `&&` and `||` in C++, and with `and` and `or` in Python.) For example, the right-hand operand of `&&` is not evaluated if the left-hand operand is false. It's quite easy to translate expressions with `&&` into Goto-C. For example,

```
if (y > 0 && x / y >= 10) {
    foo(x);
    bar(y);
}
```

can be coded as

```
if (y <= 0) goto end_if;
if (x / y < 10) goto end_if;
foo(x);
bar(y);
end_if:
```

Expressions with `||` can be coded in goto-C using a simple adjustment of the technique used for expressions with `&&`.

## What to Do

Copy the file `encm369w18lab01/exF/logical_or.c`

Read the file and follow the instructions in the comment at the top of the file.

## What to Hand In

Nothing.

## Exercise G: A more complicated program in Goto-C

### What to Do

Copy the file `encm369w18lab01/exG/exG.c`

Read the file carefully. Make an executable using the following command:

```
gcc -Wall exG.c
```

Run the executable eight times, using the following text as input:

INPUT	DESCRIPTION
1.0 x	first run, test with invalid input
1.0 0	second run, also invalid input
1.0 3	third run
1.0 4	fourth run
1.5 10	fifth run
1.6 10	sixth run
-4.3 10	seventh run
-4.4 10	eighth run

Translate the definitions of `main` and `polyval` to Goto-C. Be careful to translate *every* use of a C feature that is not allowed in Goto-C.

Test your modified program with the same input given above for the original program. Make sure it always generates *exactly* the same output as the original program.

### What to Hand In

Printouts of the final version of your source file and the outputs of all of your test runs.

## Exercise H: Nested loops in Goto-C

### What to Do

Copy the file `encm369w18lab01/exH/exH.c`

Read the file, then make an executable and run it to check what the program does.

Translate the definitions of `print_array` and `sort_array` to Goto-C. Be careful to translate *every* use of a C feature that is not allowed in Goto-C.

Make sure your modified program generates *exactly* the same output as the original program.

### What to Hand In

A printout of the final version of your source file.