Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Winter 2014

Lab 1

Attendance/Demo

To receive credit for this lab, you must make a reasonable effort to complete the exercises and demonstrate the code you complete.

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. Finish any exercises that you don't complete by the end of the lab on your own time. Also, you must submit your lab work to cuLearn by the end of the lab period. (Instructions are provided in the *Wrap Up* section at the end of this handout.)

Part 1 - Introduction to the Pelles C IDE

Objective

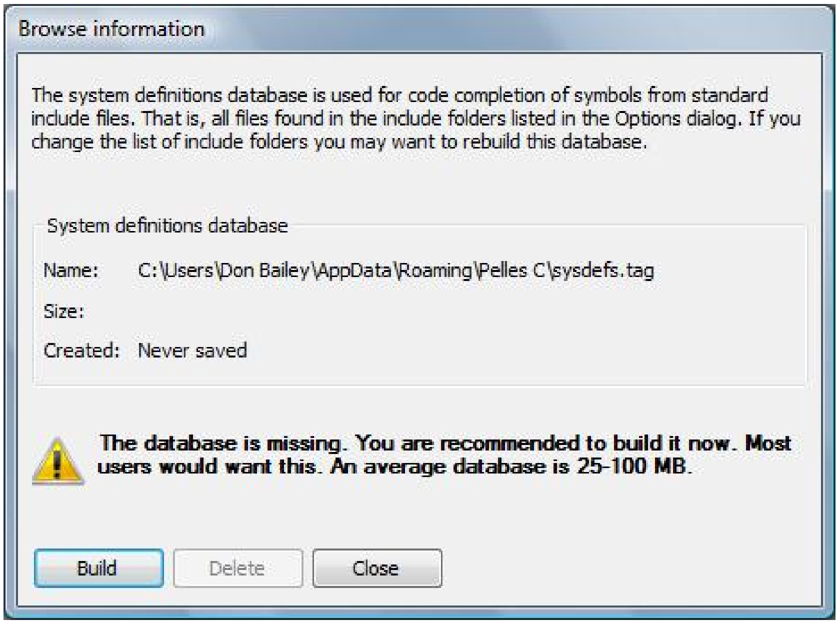
In this part of the lab, you'll become familiar with the Pelles C programming environment to prepare a simple C program. Specifically, you'll learn how to:

- create a new Pelles C project;
- create a new file containing C source code and add that file to the project;
- build the project (create an executable program);
- execute the program.

Step 1

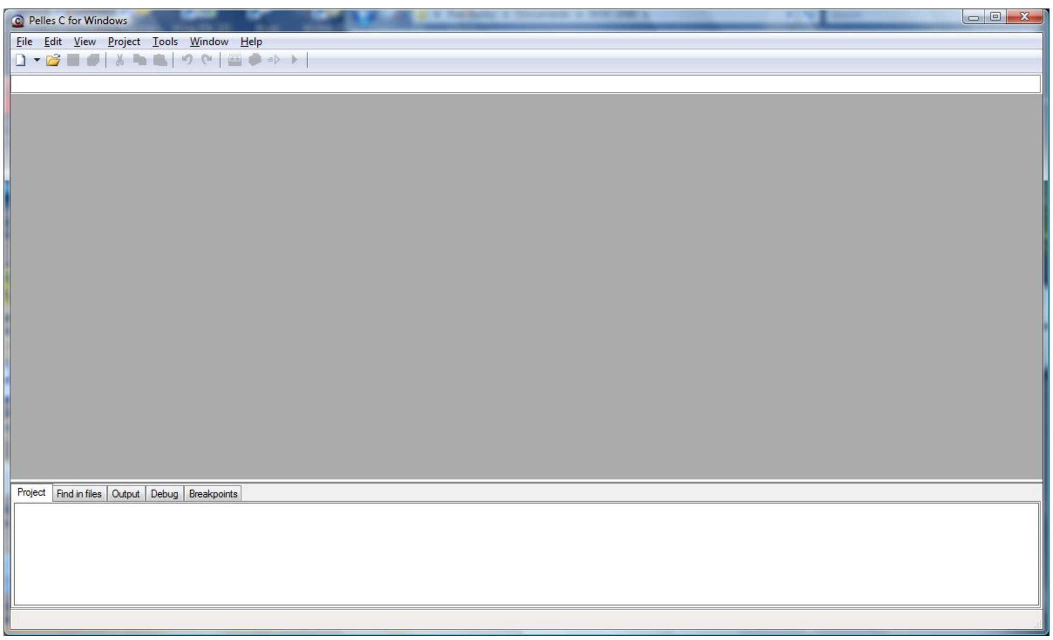
Create a folder named Lab 1.

Launch Pelles C. A Browse information dialogue box may open:



If this box appears, click the Close button.

Pelles C should now look like this:



Step 2

You're now going to create a project named hello, which will contain the classic "Hello, world!" program that's often used as the first example when learning a programming language.

From the menu bar, select File > New > Project... A New Project dialog box will appear (see the screenshot, below).

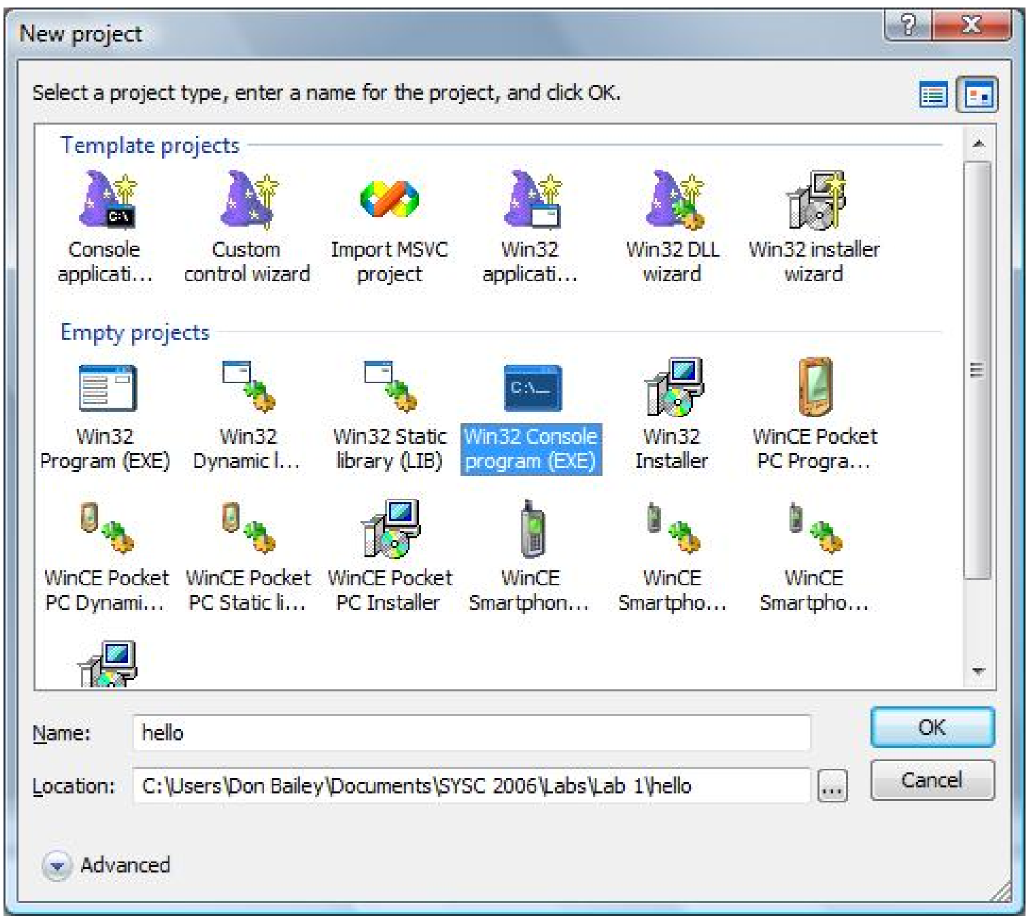
Clicking the ... button beside the Location: field allows you to navigate to the folder where you'll store your project. For example, on my computer I have a folder named SYSC 2006 that contains a folder named Labs, which in turn contains a folder named Lab 1. (This path appears in the Location field in the screenshot at the bottom of the page.)

On your computer, navigate to the Lab 1 folder you created in Step 1.

After navigating to that folder, type hello in the Name: field.

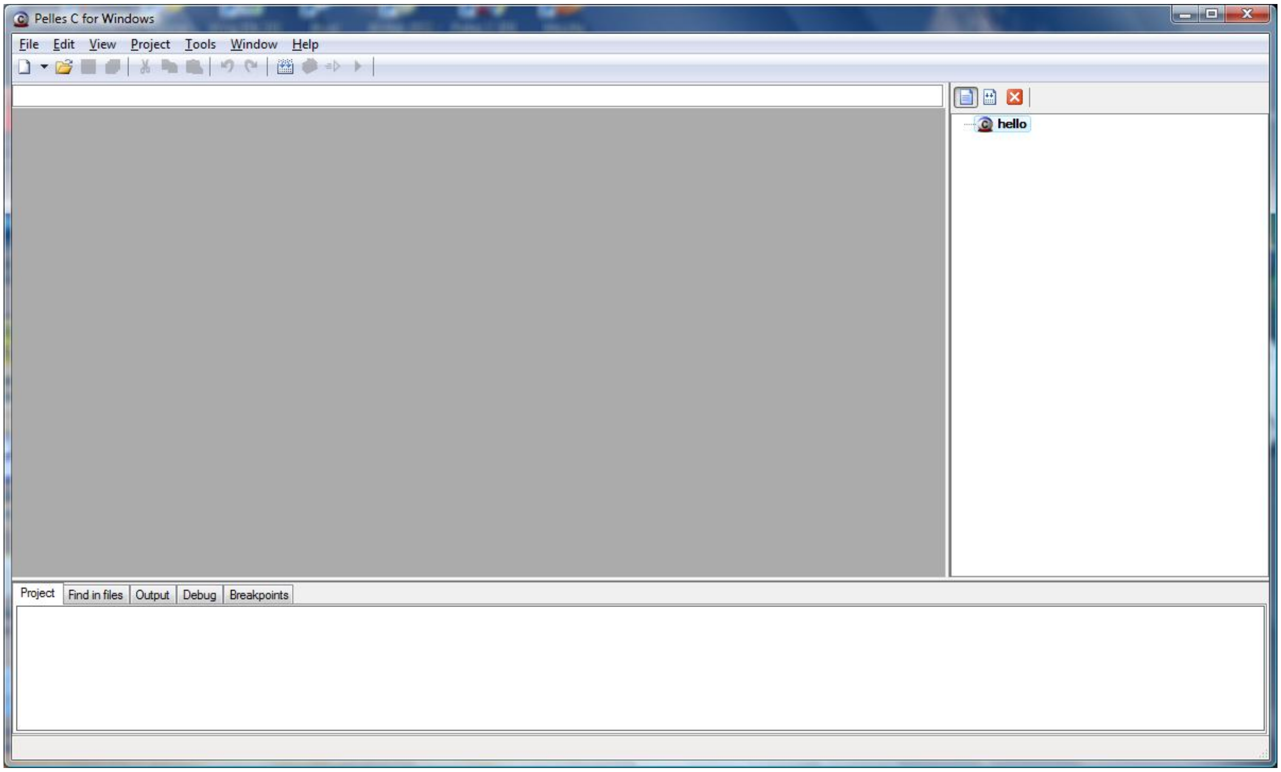
You also need to select the type of project that will be created. Click the icon labelled Win32 Console program (EXE). **Do not click Win32 Program (EXE) or any of the other empty project icons.**

Pelles C should now look like this:



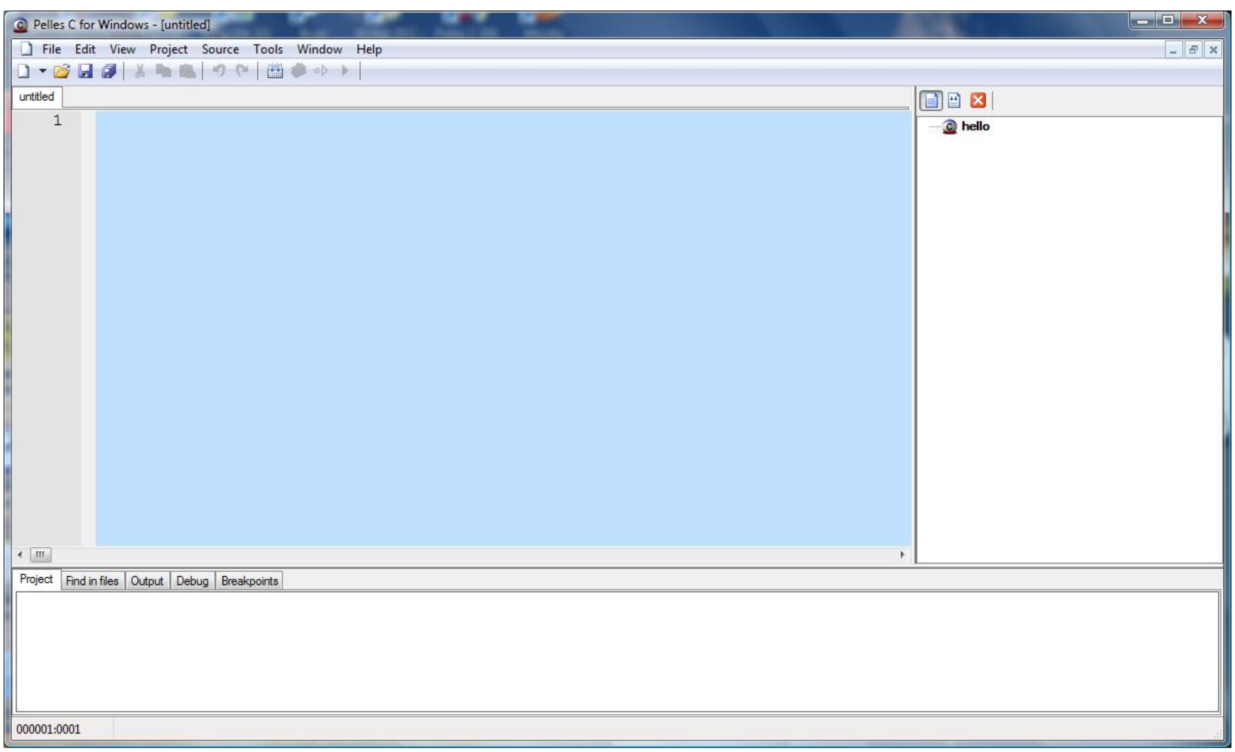
Click the OK button. Pelles C will create a folder named hello inside the Lab 1 folder. The files associated with this project will be stored there.

Pelles C should now look like this (notice that the project name, hello, now appears in the right-hand column):



Step 3

From the menu bar, select File > New > Source code. An empty blue editor window will appear:

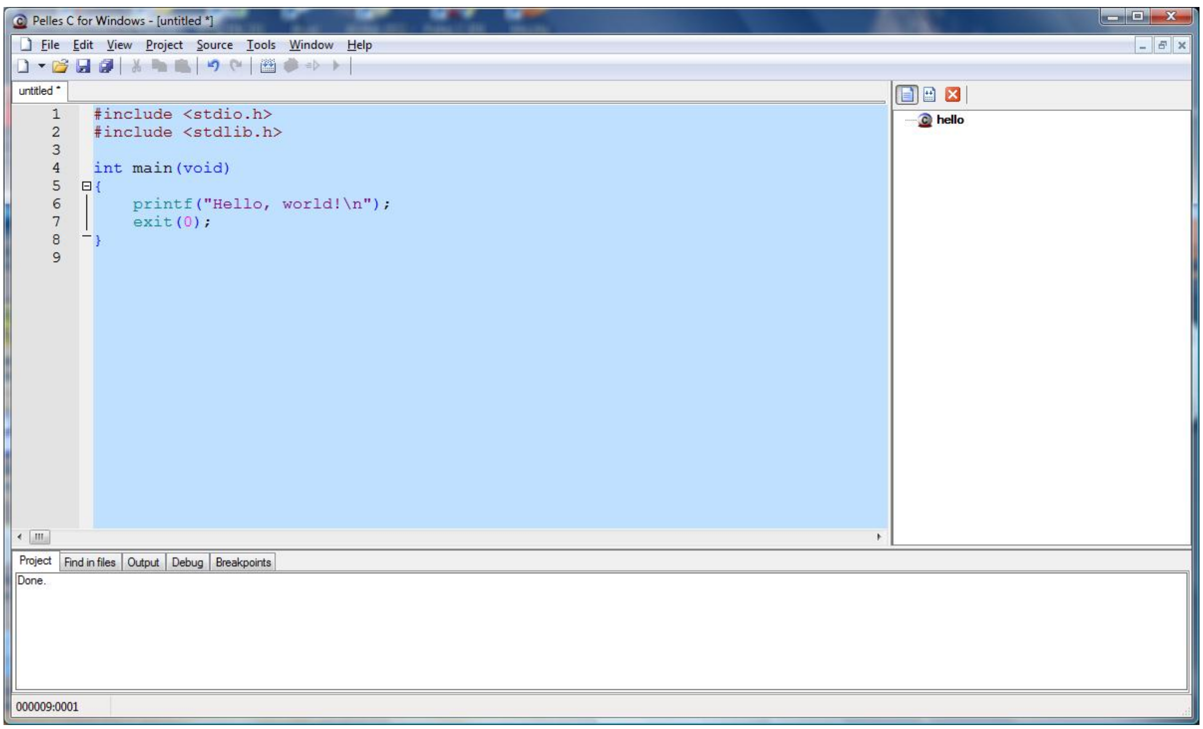


Type this C program in the editor:

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

int main(void)
{
    printf("Hello, world!\n");
    exit(0);
}
```

Pelles C should now look like this:

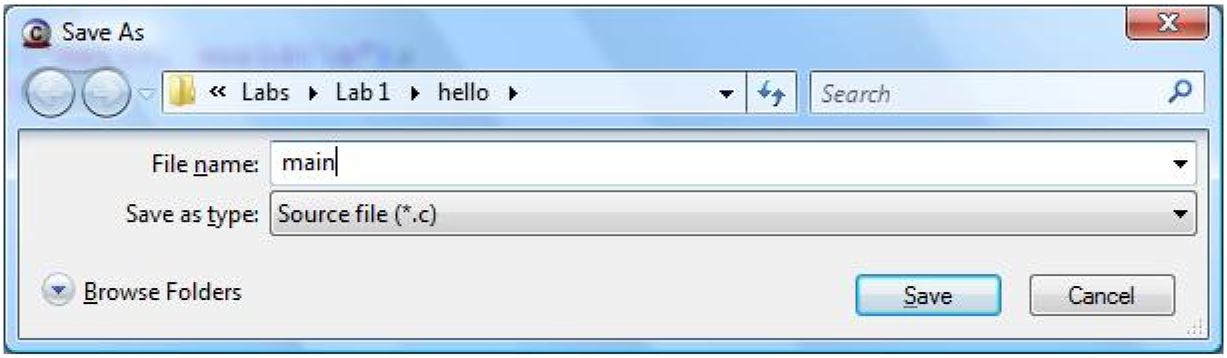


Note: the C program shown here will terminate when main calls the exit function, which is provided in C's standard library. A C program will also terminate if it executes a return statement located in the main function. The advantage of using the exit function is that it can be called from anywhere in a program, whereas return terminates a program only when it occurs inside main.

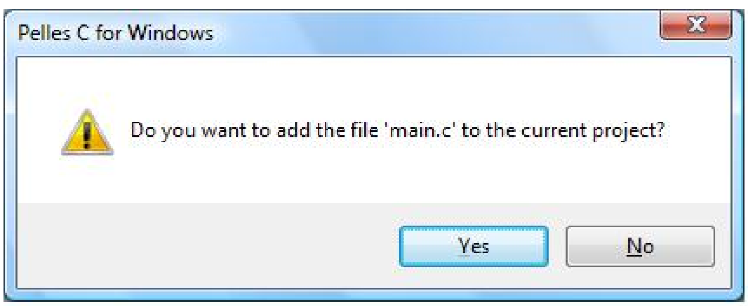
Step 4

You'll now save the source code in a file named main.c. We could choose any name for this file, but as we'll see later in the course, a C program can consist of multiple .c files, so many C programmers follow the convention of always using main.c as the name of the file that contains the main function.

From the menu bar, select File > Save as... (or or click the Save button from the row of icons below the menu bar). A Save As dialogue box will appear. Type the name main in the File name: field, and ensure that Save as type: is Source file (*.c).

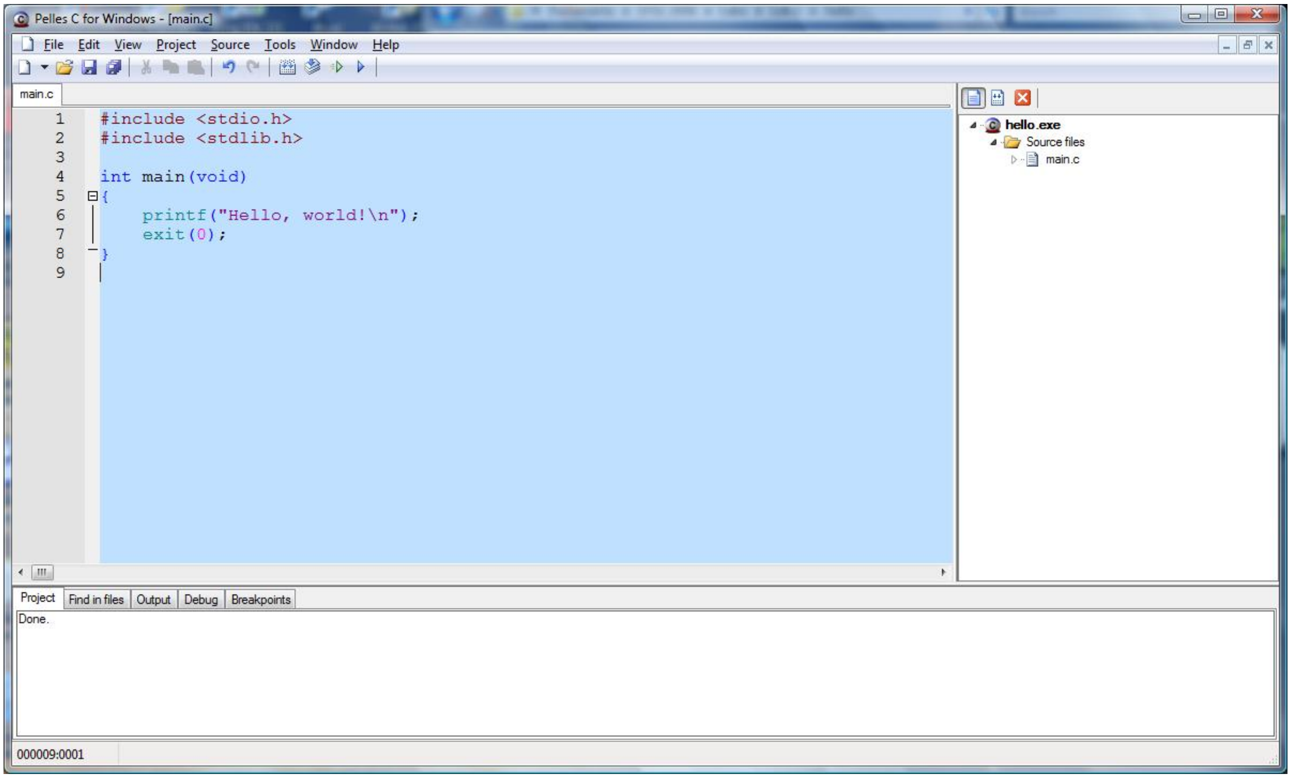


Click Save. You will now be asked if you want to add the source code file to the project:



Click Yes. Pelles C will save the code in the editor in a file named main.c and add this file to your project.

Pelles C will now look like this:



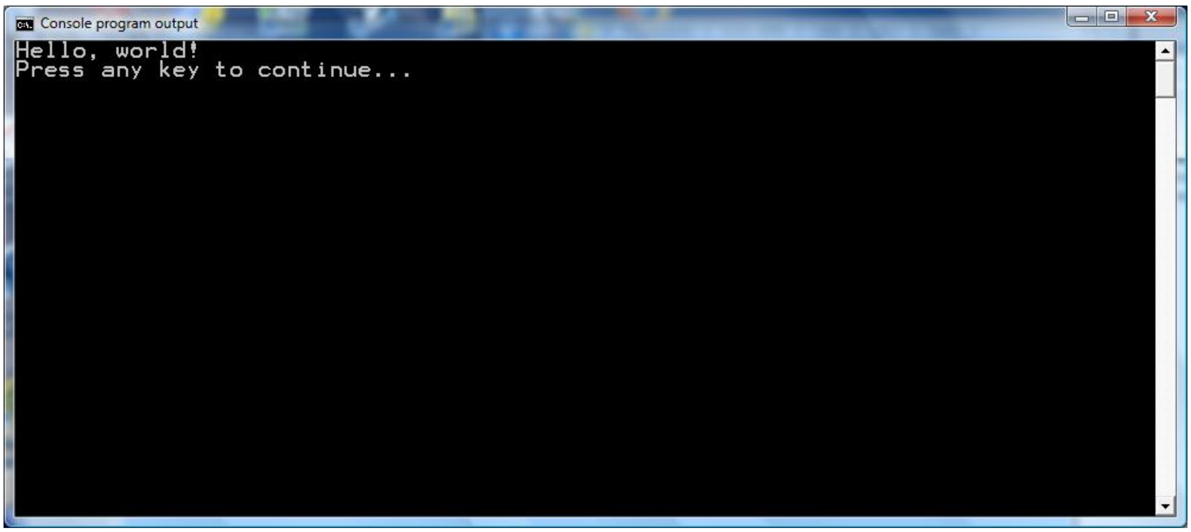
Notice that main.c is now listed as one of the source files in the hello project (see the right-hand column).

Step 5

From the menu bar, select Project > Build hello.exe (or or click the Build button from the row of icons below the menu bar). Pelles C will compile the code in main.c, and if there are no compilation errors, link the compiled code to the C libraries that the program requires, producing a file named hello.exe file (the executable program). (We will discuss compilation and linking in one of the lectures.)

Step 6

From the menu bar, select Project > Execute hello.exe (or or click the Execute button from the row of icons below the menu bar). A Console window will appear, showing the program's output:



Press any key to close the console window.

Step 7

Edit the program, adding a printf statement that outputs "C programming is fun!".

You'll need to save the modified source code. From the menu bar, select File > Save (or or click the Save button from the row of icons below the menu bar).

Build and execute the program. It should now display:

```
Hello, world!
C programming is fun!
```

Part 2 - C Functions

Objective

The objective of this part of the lab is to design, code and test some simple functions in C.

Students who took ECOR 1606 will find that this lab reviews much of the C/C++ taught in that course (pretty well everything up to but not including arrays).

Students who took SYSC 1005 wrote functions similar to these in Python. You've already learned all the programming constructs you'll require (functions, if statements, loops); the only thing that's new is that you'll use C versions of those constructs instead of Python to implement the algorithms.

General Requirements

For those students who already know C or C++: when coding your solutions, do not use arrays, structs or pointers. They aren't necessary for this lab.

None of the functions you write should produce console output; i.e., contain printf statements.

You have been provided with file main.c. This file contains incomplete implementations of four functions you have to design and code. It also contains a *test harness* (functions that will test your code, and a main function that calls these test functions). **Do not modify main or any of the test functions.**

Step 1

Create a new project named functions inside your Lab 1 folder. The project type must be Win32 Console program (EXE). **Do not create this project inside the hello folder you created in Part 1.** After creating the project, you should have a folder named functions inside your Lab 1 folder, in addition to the hello folder (check this).

Step 2

Download files main.c and sput.h from cuLearn. Move these files into your functions folder.

Step 3

You must also add main.c to your project. To do this, select Project > Add files to project... from the menu bar. In the dialogue box, select main.c, then click Open. An icon labelled main.c will appear in the Pelles C project window.

You don't need to add sput.h to the project. Pelles C will do this after you've added main.c.

Step 4

Build the project. It should build without any compilation or linking errors.

Step 5

Execute the project. The test harness will report several errors as it runs, which is what we'd expect, because you haven't started working on the functions the harness tests.

The console output will be similar to this:

```
== Entering suite #1, "Exercise 1: factorial()" ==

[1:1]  test_factorial:#1  "factorial(0) ==> 1"  FAIL
!    Type:      fail-unless
!    Condition: factorial(0) == 1
!    Line:      58
[1:2]  test_factorial:#2  "factorial(1) ==> 1"  FAIL
!    Type:      fail-unless
!    Condition: factorial(1) == 1
!    Line:      59
[1:3]  test_factorial:#3  "factorial(2) ==> 2"  FAIL
!    Type:      fail-unless
!    Condition: factorial(2) == 2
!    Line:      60
[1:4]  test_factorial:#4  "factorial(3) ==> 6"  FAIL
!    Type:      fail-unless
!    Condition: factorial(3) == 6
!    Line:      61
[1:5]  test_factorial:#5  "factorial(4) ==> 24"  FAIL
!    Type:      fail-unless
!    Condition: factorial(4) == 24
!    Line:      62

--> 5 check(s), 0 ok, 5 failed (100.00%)

== Entering suite #2, "Exercise 2: ordered_sets()" ==

...

==> 15 check(s) in 3 suite(s) finished after 1.00 second(s),
     0 succeeded, 15 failed (100.00%)

[FAILURE]
*** Process returned 1 ***
```

This term, we are going to use a test framework named sput (Simple, Portable Unit Testing framework for C/C++) . At this point in the course, we don't expect you to be able to use sput to code a test harness, but a few paragraphs will help you understand the output it produces.

File main.c contains three *test suites*, one for each of the functions you'll write in Exercises 1-3.

In Exercise 1, you'll complete the implementation of a function named factorial. The first test suite is named "Exercise 1: factorial()". This test suite has one *test function*, named test_factorial. This function calls factorial five times, to calculate 0!, 1!, 2!, 3! and 4!. Each time, the value returned by factorial is compared to the value we expect a correct implementation of the function to return.

For example, the first test performed by test_factorial checks if factorial correctly calculates 0!:

```
[1:1]  test_factorial:#1  "factorial(0) ==> 1"  FAIL
!    Type:      fail-unless
!    Condition: factorial(0) == 1
```

The condition indicates that test_factorial expects the value returned by factorial(0) to equal 1. The incomplete implementation of factorial in main.c always returns -1, so this test fails.

After the first suite has been executed, a summary is displayed, indicating that all 5 tests performed by test_factorial failed:

```
--> 5 check(s), 0 ok, 5 failed (100.00%)
```

After you have correctly implemented factorial, the output displayed by sput should look something like this:

```
== Entering suite #1, "Exercise 1: factorial()" ==

[1:1]  test_factorial:#1  "factorial(0) ==> 1"  pass
[1:2]  test_factorial:#2  "factorial(1) ==> 1"  pass
[1:3]  test_factorial:#3  "factorial(2) ==> 2"  pass
[1:4]  test_factorial:#4  "factorial(3) ==> 6"  pass
[1:5]  test_factorial:#5  "factorial(4) ==> 24"  pass

--> 5 check(s), 5 ok, 0 failed (0.00%)
```

From this, you can quickly determine that your `factorial` function passes all of the tests performed by `test_factorial`.

Step 6

Open main.c in the editor. Design and code the functions described in Exercises 1 through 4.

Exercise 1

The factorial $n!$ is defined for a positive integer n as:

$$n! = n \times (n\text{-}1) \times (n\text{-}2) \times \ldots \times 2 \times 1.$$

For example, $4! = 4 \times 3 \times 2 \times 1 = 24$.

0! is defined as: $0! = 1$.

Write a C function named factorial that has one parameter, n. The function header is:

```
int factorial(int n)
```

This function calculates and returns n!. Your function should assume that n is 0 or positive; i.e., **the function should not check if n is passed a positive or negative value.**

Aside: for C compilers that use 32-bit integers, the largest value of type int is $2^{31} - 1$. Because the return type of factorial is int and $n!$ grows rapidly as n increases, this function will be unable to calculate factorials greater than 15!

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Look at the console output, and verify that your function passes all of the tests in the first test suite before you start Exercise 2.

Exercise 2

Suppose we have a set of n distinct objects. There are $n!$ ways of ordering or arranging n objects, so we say that there are $n!$ permutations of a set of n objects. For example, there are $2! = 2$ permutations of {1, 2}: {1, 2} and {2, 1}.

If we have a set of n objects, there are $n!/(n-k)!$ different ways to select an ordered subset containing k of the objects. That is, the number of different ordered subsets, each containing k objects taken from a set of n objects, is given by:

$$n!/(n-k)!$$

For example, suppose we have the set {1, 2, 3, 4} and want an ordered subset containing 2 integers selected from this set. There are $4! / (4 - 2)! = 12$ ways to do this: {1, 2}, {1, 3}, {1, 4}, {2, 1}, {2, 3}, {2, 4}, {3, 1}, {3, 2}, {3, 4}, {4, 1}, {4, 2} and {4, 3}.

Write a C function named ordered_subsets that has two parameters, n and k, and has return type int. This function returns the number of ways an ordered subset containing k objects can be obtained from a set of n objects. Your function should assume that n and k are positive and that n >= k; i.e., the function should **not** check if n and k are passed positive or negative values, or compare n and k.

For each factorial calculation that's required, your ordered_subsets function must call the factorial function you wrote in Exercise 1. In other words, don't copy/paste code from factorial into ordered_subsets.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Look at the console output, and verify that your function passes all of the tests in the first test suite before you start Exercise 3.

Exercise 3

Combinations are not concerned with order. Given a set of *n* distinct objects, there is only one combination containing all *n* objects.

If we have a set of of *n* objects, there are $n!/(k!)(n-k)!)$ different ways to select *k* unordered objects from the set. That is, the number of combinations of *k* objects chosen from a set of *n* objects is:

$$n!/((k!)(n-k)!)$$

The number of combinations is also known as the *binomial coefficient*.

For example, suppose we have the set {1, 2, 3, 4} and want to choose 2 integers at a time from this set, without regard to order. There are 4! / (2! * (4 - 2)!) = 6 combinations: {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4} and {3, 4}.

Write a C function named binomial that has two parameters, n and k, and has return type int. This function returns the number of combinations of k objects chosen from a set of n objects. Your function should assume that n and k are positive and that n >= k; i.e., the function should **not** check if n and k are passed positive or negative values, or compare n and k.

Your binomial function must call your ordered_subsets and factorial functions.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Look at the console output, and verify that your function passes all of the tests in the first test suite before you start Exercise 3.

Exercise 4

The cosine of an angle *x* can be computed from the following infinite series:

$$\cos x = 1 - x^2/2! + x^4/4! - x^6/6! + \ldots$$

We can approximate the cosine of an angle by summing the several terms of this series.

Write a C function named cosine that has two parameters, x and n, and has return type double. This function calculates and returns the cosine of angle x by calculating the first n terms of the series. Note that x is measured in radians, not degrees. (Recall that there are Π radians in 180 degrees.) Your cosine function must call your factorial function.

Your cosine function must call C's pow function. The function prototype is in header file math.h:

```
// Return x raised to the y power.
double pow(double x, double y);
```

Note that it's o.k. to pass an integer arguments to pow. For example, if the second argument is an integer, C will convert this value to a double before assigning it to parameter y.

For this exercise, instead of using a sput test suite, we'll use a different approach to testing the function. The C standard library has a function named cos, so we'll compare the cosines calculated by this function with the values returned by your cosine function.

main.c contains a function named test_cosine. Here is the code that lets us check if cosine correctly calculates the cosine of 0 radians. It first calls C's cos function to calculate a correct approximation of cos(0). It then repeatedly calls your cosine function. The first time cosine is called, only the first term of the series is calculated. The second time cosine is called, two terms of the series are summed. During the final iteration, seven terms are summed. When you run this code and observe the output, you'll see how rapidly the value returned by cosine converges on the correct value (as returned by C's cos function).

```
printf("Calculating cosine of 0 radians\n");
printf("Calling standard library cos function: %.8f\n", cos(0));
printf("Calling cosine function\n");
for (int i = 1; i <= 7; i += 1) {
   printf("# terms = %d, result = %.8f\n", i, cosine(0, i));
}
printf("\n");
```

Notice that the character string argument in the fourth call to printf is "# terms = %d, result = %.8f\n". When this string is displayed, the %d will be replaced by the value of variable i and the %.8f will be replaced by the value returned by cosine. %.8f specifies that this value should be formatted as a double (a real number), with 8 digits after the decimal point.

The test function calculates the cosines of 0 radians (0 degrees), Π /4 radians (45 degrees), Π/2 radians (90 degrees), and Π radians (180 degrees). For each of these values, we have cosine calculate 1 term of the series, 2 terms of the series, etc., all the way up to 7 terms.

Inspect the output produced by test_cosine. How close are the values returned by cosine to the values returned by cos?

What are the advantages and disadvantages of testing your cosine function using the approach followed in this exercise, compared to using a test framework like sput?

Wrap-up

1. Remember to have a TA review and grade your solutions to the exercises before you leave the lab.

2. The next thing you'll do is package the project in a ZIP file (compressed folder). From the menu bar, select Project > ZIP Files... A Save As dialog box will appear. Click Save. Pelles C will create a compressed (zipped) folder named functions.zip, 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 functions).

3. Log in to cuLearn, click the Submit Lab 1 link and submit functions.zip. After you click the Add submission button, drag the file to the File submissions box. After the icon for the file appears in the box, click the Save changes button. At this point, the submission status for your file is "Draft (not submitted)". You can resubmit the file by clicking the Edit my submission button. After you've finished uploading your file, remember to click the Submit assignment button. This will change the submission status to "Submitted for grading".