Introduction

September 6, 2023

1 Introduction

During this course exercises will be provided for self-study to help you learn the foundations of C and C++. You are free to work through these exercises in any way you would like to, but for your convenience we have provided a set of Jupyter notebooks which can be used to interactively write, compile and execute C and C++ code in your browser.

This notebook explains the workflow for writing and executing code with these notebooks.

These notebooks are provided for ease of instruction and are not intended for any serious programming work. If you continue to use C/ C++ after this course, and *particularly if you are a MISCADA student*, It is **strongly recommended** that you install a C/C++ compiler on your own computer along with whatever editor or IDE you prefer to use. Perhaps the most notable compilers are GCC and clang.

1.0.1 Jupyter Notebooks

Jupyter Notebooks are shareable documents combining text, code and rich multimedia output which are very useful for interactive computing. A notebook is just a series of ``cells'', each of which contains code or text (in markdown form). You can change the type of a cell with the dropdown menu at the top of the page, shown here:

Here are some useful keyboard shortcuts for using notebooks:

To:	Use:
Navigate between cells	Up & down keys
Edit a cell	`Enter`
Leave a cell you are editing	`Esc`
Execute a cell	`Ctrl+Enter`
Execute a cell and select next cell	`Shift+Enter`
Execute a cell and insert a cell after	`Alt+Enter`

1.0.2 Kernels

A notebook is nothing more than a document combining text, code and possibly multimedia. The notebook sends all requests for computation to a kernel working in the background. You can see the kernel this notebook is using in the top-right corner of the screen alongside an icon indicating its status. This kernel is a custom one developed to make it easier to learn C and C++ in your

browser. It allows you to write C and C++ code in a notebook and automatically compiles and runs it for you, showing you the commands used to do so.

Note: If the kernel becomes unresponsive you may have to restart it by going to **Kernel** -> **Restart Kernel**.

The kernel used by this notebook is available here if you want to try it out locally.

2 Working with code

2.0.1 Editing code

By default, new cells are code cells. What happens when you execute the code cell below?

[ ]:
```c
#include <stdio.h>

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

That code cell didn't work because code cells have to be saved before they can be compiled.

The kernel used by this notebook requires that the first line of each code cell should be a comment starting with `////` followed by a unique filename (with either the `.c` or `.cpp` extension). The code in the cell will then be saved and compiled as either C or C++.

It is also recommended to turn on line numbers by going to **View** -> **Show Line Numbers**.

2.0.2 Compiling & running code

Let's see what is required to create a simple ``hello world'' program in C. Below is the source code for our program. Execute the code cell to save it to the file `hello-world.c`:

Note: The `%%file` on the first line is a ``magic'' command to the kernel to save the cell contents to a file. Later we'll see more examples of other magic commands…

[ ]:
```c
%%file hello-world.c
#include <stdio.h>

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

The cell below contains a single shell command which lists all files ending with `.c`. Execute it to see that our file `hello-world.c` was created.

Note: cells with a single line starting with `!` are interpreted as shell commands.

[ ]:
```
! ls
```

To turn our source file into an executable, we need to use a compiler. The command below uses `gcc` to compile the source file `hello-world.c` into an executable `hello-world`. Execute this cell to compile the program:

[ ]:
```
! gcc hello-world.c -o hello-world
```

Now list the files in the current directory to see that our executable was created:

[ ]:
```
! ls
```

Now run the program:

[ ]:
```
! ./hello-world
```

2.0.3 Compiling & running code automatically

The kernel we are using with these notebooks is capable of taking care of compiling and executing our programs for us automatically, so that you can focus on just learning & writing C/C++. Let's see how to do this.

The code cell below contains the same source code as above, but the first line is different. Code cells which start with `////` followed by a file name are interpreted as source code which should be saved in the specified file. The notebook kernel saves the code in the file and then invokes the compiler, showing you the commands used to do so. It automatically executes the program too, unless you tell it not to. Execute the code cell below to see this in action:

Note: add the `//% NOEXEC` magic comment after the first line to prevent automatic execution of the program.

For a list of magic comments that can be used in code cells, see this link.

[ ]:
```
//// hello-world.c
#include <stdio.h>

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

The output below the code should look like this:

```
wrote file hello-world.c
$> gcc    hello-world.c -o hello-world
$> ./hello-world
hello, world!
```

This output shows us that the kernel has saved the code in the file `hello-world.c` and has then compiled that file into a program called `hello-world` using the command:

```
gcc hello-world.c -o hello-world
```

Then the program has been executed with:

```
./hello-world
```

The output from the program is then shown below this command.

This code cell does the same in C++:

[ ]:
```
//// hello-world-cpp.cpp
#include <iostream>

int main(){
    std::cout << "hello, world!\n";
    return 0;
}
```

2.1 Adding compilation options

When compiling C/C++ it is often necessary to change the compiler command with various options and flags. As an example, execute the code cell below to save the code to a file and then run the compiler command in the next cell. Add the flags `-Wall` and `-Werror` to the command to see a difference. These flags turn on many useful warnings (`-Wall`) and promote all warnings to errors (`-Werror`).

Note: Having these flags enabled will help you spot many programming errors while learning C/C++. A **vast** array of compiler options are available. For a list of GCC options, see this page.

[ ]:
```
%%file options.c
#include <stdio.h>

int main(void) {
    int k=7;
    return 0;
}
```

[ ]:
```
! gcc options.c -o options
```

As the kernel we are using here invokes the compiler for us automatically, we can add options and flags to the command using this special comment in a code cell:

```
//% CFLAGS [options and flags ...]
```

Execute this code cell to see what the result is:

[ ]:
```
//// options.c
//% CFLAGS -Wall -Werror
#include <stdio.h>

int main(void) {
    int k=7;
    return 0;
}
```