Getting started with scientific programming

This course pack is geared toward getting someone without any programming experience up and running with scientific programming. For this course, we will use the C programming language. An alternative language that is often used for scientific application is Fortran. C and Fortran are actually quite similar, and if you learn one well, the other is pretty simple to pick up. In fact, many scientific programmers are versatile in both languages.

To get started, you will need to have a compiler installed on the computer you plan to use. If you're using a linux workstation, you should be set to go already. If you're using a Mac, you may need to download a free application package. If you're using a PC, one solution is to download the CYGWIN program, which provides a linux style pseudo-operating system on your PC. There are several commercial compilers out there, but in this course, we will use the free-ware compiler developed by GNU.

In the most simple manner, a computer code requires 2 things: the source code and an executable file. The source code is a file (or multiple files linked together) that is written in the programming language. In our case, the source code will be a text file that contains instructions written in the C programming language. We use a program called a "compiler" to convert our written instructions into a machine code that the computer can understand. The resultant machine code is called the executable file. One runs the executable file to run their program. A note about portability: As long as you write your source code in standard C language, you should be able to compile it on any machine. However, the executable file that your compiler produces is usually machine-specific, meaning that it will only run on the same type of machine that it was compiled on. Therefore, it is important to keep in mind that it is your source code that is most valuable. As long as you have it, you can compile it on any machine you choose. But don't try to copy your executable to another machine and run it. Even if it seems to work, there could be unintended consequences behind the scenes. When moving to a new machine, always recompile your source code on that new machine to create a new executable file.

As mentioned, there are many compilers on the market today, but most scientists use the ones created by the freeware GNU project. In this course, we will use the GNU compiler exclusively. Note that the GNU compiler is often just called the "gcc compiler". One thing to note is that some compilers (or different versions of the same compiler) may have different levels of pickiness with the style of code you write. We will write completely in the standard C language, so all compilers should be able to understand our codes; however, different compilers may produce different warning messages. For example, the compiler on our server may give different warnings than the compiler on your local machine.

As you will see, it is a very rare occasion indeed that you write a code and then compile it without problems. It is common to make all kinds of little mistakes when writing a code. Most of the time (surely not always though!), the compiler will find these mistakes for us. The compiler will issue two types of statements to us when complaining about our source code. One type is a "warning" and the other, more serious type is an "error".

A warning is a potential problem with the code that is not bad enough to stop the compiling process. In other words, the compiler may give you many warnings, but it will still compile your code anyways. A very bad habit is to ignore warnings. To promote good code writing behavior, we will insist that your code produces no warnings during the compiler process. Sometimes, you will notice that a code that produces no warnings on machine A will produce warnings on machine B. This means that the compiler on machine B is more stringent. It is good practice to fix the warnings produced by the more stringent compiler in this case.

An error is a problem deemed bad enough to stop the compiling process altogether. It means that there is some fatal flaw in your source code, and the compiler cannot translate it to machine code. You are forced to fix this error before you can compile. It is probably worthwhile to bring up a cautionary note here. It is good practice to compile often as you write code. Therefore, there is usually a pre-existing executable file in your directory. If the compiler produces an error and you don't notice it on the screen, you may mistakenly assume the code compiled and ignorantly just run your old executable code, thinking everything is going well. When you compile, it is important to watch the screen to see if any error messages come up. If they do, your code did not compile.

A final note before we get started. Please get into the habit of compiling very often as you work on your code. After you write a few lines of code, save your code file and compile it. One of the bigger mistakes a beginner makes is to write an entire code without compiling it at all throughout the writing process. What typically happens is that when you finally compile it, it contains so many errors and warnings that it is impossible to debug! Also, put many print statements in your code as you develop it to make sure that variables contain the numbers they should be containing.

Also, C is case-sensitive, meaning *a* is different than A.

Compiling your first code

Every C code has a "skeleton" structure that you will usually start with. Let's make one here. Open a text file and name it *code01.c* . Write the following text in the file:

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

int main(void)
{


return 0;
}
```

The first 3 lines tell the compiler to include 3 “standard” libraries. Standard libraries are included with every compiler. We could also write our own libraries if we wanted to. A library consists of code that has already been written for us; these are usually functions that have specific purposes. For example, *stdio.h* contains many functions related to input and output, *stdlib.h* contains many general purpose functions such as memory allocation, and *math.h* contains mathematical functions that we will use quite often in this course. Even though we don’t always need these 3 libraries, it is a good habit to always include them in every code.

After the three include statements is the main function. Every program starts in the main function. Don’t worry about the *int* and *void* statements; they may make more sense later. The *return 0* line tells the code to return the integer value 0 after the code has finished.

Let’s compile your code. First, make sure your text file is saved and you have exited from the text editor. To compile, type the following on the command line.

```
gcc –lm code01.c
```

The *–lm* is a flag to instruct the compiler to include the math library. You only need it if you use mathematical functions. We don’t in this case, but let’s get into the habit of including the *–lm* flag.

If the compiler was successful, then it won’t produce any text whatsoever. That’s a good thing; it means your code compiled without a hitch. If you list your directory, you will notice that a new code appeared. It is named *a.out* (or if you are using CYGWIN, it is named *a.exe*). Now, let’s run our executable by typing:

a.out OR **a.exe** if you are using CYGWIN.

If you get an error saying that “command not found”, then it likely means that the current directory is not in your path. The instructor will help you set this up, but in the meanwhile, simply type the following (which tells your OS to look for the executable in your current directory).

./a.out OR **./a.exe** if you are using CYGWIN

Ok, you ran your code, but it didn't do anything, right? That's because we didn't provide it with any instructions. Let's make the code say Hello. Open the file *code01.c* and add the following line in the main function, somewhere between the first bracket and the return line.

```
fprintf(stdout,"Hello World\n");
```

The *fprintf* command is used to print things. Here, we are telling the code to print to standard screen output, which is denoted by the *stdout* term. It will write whatever is in the following quotes. Note, there must be a comma before the quoted part. The *\n* term tells the code to move to a new line. In other words, *\n* is a carriage return instruction. Don't forget to put the semicolon at the end of the instruction. In C, semicolon symbolizes the end of the instruction. A common mistake in the beginning is to forget putting semicolons at the end of an instruction.

So, your text should now look like this:

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

int main(void)
{

fprintf(stdout,"Hello World\n");

return 0;
}
```

Compile your code and then run the executable. If successful, you will see the words *Hello World* appear on your screen.

Just for fun (and to familiarize yourself with a very common compiler warning), go back to the source code and remove the semicolon after the fprintf statement. Notice the resultant error when you do so. Get used to this error statement because you will likely forget semicolons in the beginning. Now go back and fix the error.

If you don't like your executable file having the name *a.out*, you can change it by including the output flag when you compile. For example:

```
gcc –lm –o mycode.x mycode.c
```

The *–o mycode.x* part tells the compiler to produce an executable named *mycode.x* instead of *a.out.* Be careful though because if you accidently typed *gcc –lm –o mycode.c mycode.c* instead, your source code would get wiped out!!!

Working with Data Types (i.e., variables)

In C, there are numerous data types. The most common are **integers**, **floats**, and **doubles**. Integers are simply integers, floats are relatively low-precision real numbers, and doubles are high-precision real numbers. In the past, when memory was expensive, one would try to use floats instead of doubles to save space in RAM. However, floats only have about ~5-6 digit precision (how many digits is actually machine-dependent). In this class, we will not use floats at all, and rely totally on doubles for real numbers.

Make a new text file named *code02.c* and write the following source code:

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

int main(void)
{

// This is what a comment looks like.  Any text after two dashed lines is
// ignored by the compiler.

// The first part of a code involves the declaration of variables.

// beginning of declarations
int number_of_apples, number_of_oranges;
int number_of_people;
int total_number_of_fruit;

double apples_per_person;
double oranges_per_person;

// end of declarations

// lets add some values to variables

    number_of_apples=20;
    number_of_oranges=15;
    number_of_people=12;

// lets do some math

    total_number_of_fruit = number_of_apples + number_of_oranges;

    apples_per_person = number_of_apples/number_of_people;
    oranges_per_person =
          (number_of_oranges*1.0)/(number_of_people*1.0);

// lets print some results

    fprintf(stdout," There are %d pieces of fruit total\n\n", total_number_of_fruit);

    fprintf(stdout," There are %f apples per person\n", apples_per_person);
    fprintf(stdout," There are %f oranges per person\n", oranges_per_person);
```

```
   return 0;
   }
```

Compile and run your code. You output should look like this:

```
There are 35 pieces of fruit total

 There are 1.000000 apples per person
 There are 1.250000 oranges per person
```

Do you notice that there is something wrong with the math? Ok, we'll cover that in a bit, but first let's go over your code first.

First of all, it is important to add comments to code, but be careful not to over-comment, which can actually cause more of a hindrance than help. A comment is denoted by two slashes, //. Any text on the same line AFTER the slashes is ignored by the compiler.

Before actual instructions are written, you must declare your variables. Variables can have any name EXCEPT the following words which are reserved for more important things:

```
auto, break, case, char, const, continue, default, do, double, else, enum,
extern, float, for, goto, if, int, long, register, return, short, signed, sizeof,
static, struct, switch, typedef, union, unsigned, void, volatile
```

It is good form to make your variable names mean something. This allows commenting to be minimized. Also, a common practice is to name integer variables something that starts with the letters I through Q, and real variables starting with something else.

Before you provide the variable name, you must indicate what type of variable it is. In our example problem, the variables are either **double** or **integer**. Note, that an integer is declared as only **int**. On a single line, you can declare as many variables of a given type as you would like. For example,

```
int number_of_apples, number_of_oranges;
```

is equivalent to:

```
int number_of_apples;
int number_of_oranges;
```

Another thing to point out here; NEVER do the following:

```
number_of_apples = 20.0
```

because in this case, *number_of_apples* has been declared as an integer and you are trying to give it a real number (i.e., something with decimal digits). This could cause some strange problems!

Lets take a closer look at one of the print statements that we included.

```
    fprintf(stdout," There are %f apples per person\n", apples_per_person);
```

Within the quoted portion, the *%* sign indicates a placeholder for a particular variable value. In this case, *%f* is a placeholder for a real number. In the other print statement, we used *%d* which is a placeholder for an integer. After the quoted portion, you would put the variables that are to be printed in the placeholders. The variables need to be separated by commas.

Now, back to our math problem. The number of oranges per person was correct, but the number of apples per person surely wasn't. What is wrong here? This is something that you should always be careful to avoid – integer division. We'll see later that integer division can be very useful, but it this case it provides the wrong answer. Basically, **when you do math with integers, the answer is always an integer**. You should always multiply an integer by 1.0 if you want to do math with it.

Go back to your source code and make the following correction:

```
  apples_per_person = (number_of_apples*1.0)/(number_of_people*1.0);
```

Recompile and run. You should now get the following correct answer:

```
     There are 35 pieces of fruit total

      There are 1.666667 apples per person
      There are 1.250000 oranges per person
```

Let's revisit the print statement again. Let's say we want our output to be limited to 3 decimal places. We could do that by putting a *.3* within the placeholders as such:

```
    fprintf(stdout," There are %.3f apples per person\n", apples_per_person);
    fprintf(stdout," There are %.3f oranges per person\n", oranges_per_person);
```

We now get:

```
     There are 1.667 apples per person
     There are 1.250 oranges per person
```

We could also use scientific notation with 5 decimal places by the following:

```
        fprintf(stdout," There are %.5e apples per person\n", apples_per_person);
```

which would lead to:

```
            There are 1.66667e+00 apples per person
```

Search online, and you'll find many different formatting options for your output.

You can have as many placeholders within a print statement as you like. For example, try the following line:

```
   fprintf(stdout,"%f apples and %f oranges of the %d pieces\n", apples_per_person,
oranges_per_person, total_number_of_fruit);
```

Note: you do not need to wrap the text, like shown in the above line. In C, you can make each line as long as you would like (or as wide as your screen). I have to wrap text in these tutorials so they will fit within the page margins of this coursepack.

Conditionals

Very often, we want our codes to make choices. To do this, we use *conditionals*. The following table provides the basic conditionals in C.

Expression	**What it means**
>	GREATER THAN
>=	GREATER THAN OR EQUAL TO
<	LESS THAN
<=	LESS THAN OR EQUAL TO
==	EQUAL TO
!=	NOT EQUAL TO
&&	AND
||	OR

Usually, conditionals are used in *IF* statements. It's probably best to learn by example in the following tutorials.

Start a new text file named *code03.c*. This one will examine IF/ELSE statements. It is important to practice these by making some of your own codes.

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

int main(void)
{

//you can give variables a value as you declare them

double A=1.0;
double B=2.0;
double C=3.0;
double D=4.0;
double E=5.0;
double F=6.0;

// Here is a simple, one line IF statment.
// Notice the semicolon indicates that the
// IF statement is finished.  Only the one line
// IF statements require a semicolon at the end.

   if (B > A) fprintf(stdout,"%f is bigger than %f\n", B,A);


// We can add multiple commands to our IF by putting
// them in brackets. In this example, everything within
// brackets will be done if B is greater than A.

    if (B>A)
    {
      fprintf(stdout,"B is bigger than A.\n");
      fprintf(stdout,"Thats pretty cool.\n");
      fprintf(stdout,"I think A should be +1 bigger though\n");

      // I change the value of A here to make it equal to itself
      // plus 1.

       A = A + 1;

       fprintf(stdout,"Now A is %f\n", A);
    }

// lets create three blank lines in the output

   fprintf(stdout,"\n\n\n");

// We can also tell our code to do something if the conditional
// is not true.  We do this by an ELSE.

    if (B>C)
    {
      // Because B is not bigger than C,
      // this should never happen.  Lets put
      // an error message here and tell the
      // code to exit if it does happen.

      fprintf(stdout,"You should never be here!\n");
      exit(10);

      // exit(10) tells the code to quit.
    }
    else
    {
      fprintf(stdout,"Cool, I knew C was bigger\n");
    }
// We can make a more complicated structure by including an ELSE IF

    if (C>F)
    {
      fprintf(stdout,"We should not be here\n");
      exit(10);
    }
    else if (C>E)
    {
      fprintf(stdout,"We should not be here either\n");
      exit(10);
    }
    else if (C>D)
    {
      fprintf(stdout,"We should not be here either\n");
      exit(10);
    }
    else
    {
      fprintf(stdout,"You know what? I give up.\n");
    }

 // We can also nest IF statements.
 // Always be careful with your brackets

    if (B<F)
    {
      fprintf(stdout,"B is smaller than F\n");
      if (B<E)
      {
         fprintf(stdout,"B is also smaller than E\n");
         if (B<D)
         {
           fprintf(stdout,"B is also smaller than D\n");
           if (B<C)
           {
              fprintf(stdout,"B is also smaller than C\n");
           }
         }
      }
    }


 return 0;
 }
```

You should get the following output:

```
2.000000 is bigger than 1.000000
B is bigger than A.
Thats pretty cool.
I think A should be +1 bigger though
Now A is 2.000000



Cool, I knew C was bigger
You know what? I give up.
B is smaller than F
B is also smaller than E
B is also smaller than D
B is also smaller than C
```

You should now begin to write your own programs to try out IF/ELSE statements. It's best to try things out and see if they work. Try other conditionals and more complicated nesting structures.

FOR Loops

Looping is an asset to any numerical code. Oftentimes, we need to split a function up into a zillion pieces and work with each piece. One way to make a loop in C is by using *for*. For example, the following lines:

```
int number;
for (number=1; number <= 20; number=number+1)
{
    Stuff….
}
```

The above *for* loop performs stuff in the brackets 20 times. The *for* statement includes 3 components within parentheses. Each component is separated by a semicolon.

Component 1: provides a starting value for the variable listed.

Component 2: provides the condition by which the loop will end. In this case, the loop will continue as long as the integer *number* is less than or equal to 20.

Component 3: tells the integer variable *number* to increase itself by 1 each time the loop completes a single pass.

Try the following code, *code04.c*

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

int main(void)
{

int kk;

for (kk=1; kk <= 10; kk=kk+1)
{
   fprintf(stdout,"the value of kk is %d\n",kk);
}

fprintf(stdout,"Another loop\n");

for (kk=25; kk > 14; kk=kk-1)
{
   fprintf(stdout,"the value of kk is %d\n",kk);
}

fprintf(stdout,"Another loop\n");

// kk=kk+1 can also be represented as simply kk++
// so we could also write this:

for (kk=1; kk <= 5; kk++)
{
    fprintf(stdout,"the value of kk is %d\n",kk);
}

fprintf(stdout,"Another loop\n");

// kk=kk-1 can also be represented as kk--
// so we could also write this:

for (kk=1; kk > -3; kk--)
{
    fprintf(stdout,"the value of kk is %d\n",kk);
}

return 0;
}
```

You should get the following output:

```
the value of kk is 1
the value of kk is 2
the value of kk is 3
the value of kk is 4
the value of kk is 5
the value of kk is 6
the value of kk is 7
the value of kk is 8
the value of kk is 9
the value of kk is 10
Another loop
the value of kk is 25
the value of kk is 24
the value of kk is 23
the value of kk is 22
the value of kk is 21
the value of kk is 20
the value of kk is 19
the value of kk is 18
the value of kk is 17
the value of kk is 16
the value of kk is 15
Another loop
the value of kk is 1
the value of kk is 2
the value of kk is 3
the value of kk is 4
the value of kk is 5
Another loop
the value of kk is 1
the value of kk is 0
the value of kk is -1
the value of kk is -2
```