Exercise C1: Getting started with C

Computers

We strongly suggest that you use a Linux computer for this course.

Macbooks might be ok since the operating system is very close to Linux. However, none of the teaching assistants have knowledge of specifics about Macbooks, meaning that we might not be able to help you. We are using some lesser known software that might not be too difficult to install. (No promises though.)

As for Windows, we will not be able to help you at all with the installations or the setup. Therefore, if you decide to go with Windows against our recommendation you will have to figure out how to install the software that we are using and how to get set up in your own time. This means that valuable time might be lost that you will very likely need for the assignments. Thus, we strongly advice against using Windows, however we will not force you to use Linux.

Introduction

The purpose of this assignment is for you to get familiar with the C programming language. Learning C can seem like a daunting task, but we think it is a rewarding experience. The C language is very close to the hardware, and therefore, if your code is decent, it will run fast. Furthermore, you will get a brief insight into how a computer (or rather the operating system) is working in addition to the speed it provides. Therefore, in this assignment you will have to run a couple of programs and interact with them in order to get a feeling for how C works. In the end of the lab talk to one of the teaching assistants and tell them what you have understood and what is still a mystery (or unclear).

It is often preferable to use a terminal when coding in C. This is because the terminal has a lot of useful tools for C. Here are some useful commands:

- cd - This command changes directory (change directory).
- mkdir - This command creates a directory (make directory).
- ls - This command lists the file in the current directory (list).
- pwd - This command prints the current directory (print working directory).
- touch - This command creates an empty file, or updates the timestamp on an existing file.
- make - This command looks for and runs a file named "GNUMakefile", "Makefile" or "makefile" which has recepies on how to compile your code.
- gcc - This is the compiler that you will use throughout the course.
- gdb - This is a debugger for C.
- valgrind - This is a memory sanitizer for C.

A compiler is a program that translates your code into instructions readable by a computer. This means that is in a way similar to a Python interpreter, but with the big difference that the Python interpreter does the translation on the fly. The advantage of compilation is that it can make tailor-made adaptations to your particular machine, making the code run faster. A debugger is a program that lets you step through your program and see the inspect the program at different stages. There will be an exercise on the GNU Debugger, GDB, this later in the course. Lastly, since the C programming language is rather close to the hardware you have to handle the lifetime and allocation of your memory by yourself. This means that it is easy to make a mistake with your memory, which can be caught by memory sanitizers like Valgrind. We will have an exercise on a more lightweight memory sanitizer later in the course.

Types

Here is a short list of useful types for you in this course:

- int - Integer which is (most likely) 4 bytes and can store values between (-2147483648, 2147483647)
- short int - Integer with (most likely) 2 bytes
- long int - Integer with (most likely) 8 bytes
- float - decmial number with (most likely) 4 bytes
- double - decmial number with (most likely) 8 bytes
- long double - decmial number with (most likely) 16 bytes

The number of bytes of the type can be checked by the following code:

```
printf("%li\n", sizeof(int));
```

A word of warning, the precision of arithmetic operations on floats can be atrociously bad, however, the speed compared to double is often faster. We invite you to test the accuracy of float and double. This is not a course on writing good performing code so you can simply use double instead of float without thinking about it.

C is a statically typed language, which means that when a new variable is created (declared) it must be bound to a specific type, e.g. int or double. If a value is then assigned to the variable (initialisation), the type of the value must match the declared type. Converting between different types is referred to as type casting, and may be done implicitly by the compiler in certain situations. Given two integers m = 2, n = 5 and a double x = 5.6, try to deduce the results (type and value) of the program “types.c”.

Compile and run the program:

```
gcc -O0 types.c -o types && ./types
```

Pointers

Pointers is a special type of variable that contains the location of another variable in the computer's memory (i.e., the address of that variable). When reading code with pointers it might sometimes help you to read the code backwards. For example, the first line in the code block below should be read as "line is a pointer to an integer".

```
int *line;
```

The dereference operator, &, is used to get the address of a variable.

```
int b = 2;
int *c = &b;
```

Again, using the backwards reading for the third line: "c is a pointer to an integer that stores the value of b". In a sense & and * are inverse operators of each other. Run the program "pointers.c" and reflect on the usefulness of pointers. Try to get a feeling for pointers by modifying the code. You can e.g. explore the concept of pointers to pointers which will be useful later when you need to allocate matrices.

Compile and run the program by typing:

```
gcc -O0 pointers.c -o pointers && ./pointers
```

Memory layout

The RAM (random access memory) is split into two segments in your C program, the stack and the heap. We will give you a very short note on the difference between the two memory segments here. The heap is dynamically allocated during the run time of you program. This means that you can modify the size of your arrays allocated on the heap. On the other hand, you can not modify the size of your arrays on the stack during the run time of your program. The reason for this is that the stack is allocated during the start of the program. Note that the size of the stack is significantly smaller than the heap (which is limited to the size of your RAM). Linux provides a tool to check the size of the stack:

```
ulimit -a
```

A fairly common size is 8192 kB, but can be changed.

Stack

Your task in this exercise is to get the program "stack.c" to crash by increasing the size of the array. You'll quickly notice that the stack size is very limited! Furthermore, there are no checks in place to check that the stack size that you are requesting in line with what the system is providing, hence the crash. This will therefore require a bit of a trial and error for you. Try to get familiar with what is a reasonable size for the stack, don't just make a very large array and call it a day. Explore! If your program is crashing in the future, this should probably be one of the first things that you check. This is a rather common mistake, and happens to the best of us.

Compile and run the program:

```
gcc -O0 stack.c -o stack && ./stack
```

Heap

In this task, try to increase the size of the array and see what happens. You will notice that the program never really crashes. However, something is not really working as intended. Ponder on why it fails for too large arrays. You have the error message from C as a help.

Compile and run the program:

```
gcc -O0 heap.c -o heap  && ./heap
```

Scopes

Try to understand why the program won't compile, it has to do with scopes. A scope is defined by curly braces, {}. Therefore as you will see later in the course, functions, for loops, etc are defined in scopes. See: https://en.wikipedia.org/wiki/Scope_(computer_science) for reference. Your task in this exercise is to get the program to compile and run. However, don't just remove the scope or move the printf function. A hint is that you it has to do with declaration of b and parent scopes.

Compile and run the program:

```
gcc -O0 scopes.c -o scopes && ./scopes
```

Functions

Take a look at the file "functions.c". The anatomy of a function in C is as follows:

prototypes

A function prototype is something that tells the compiler what the input is to the function and what it returns. This is needed when you call a function before it is defined, but is probably good practice to include even if it is not needed. In larger projects, these are often put into so-called header files which have the file extension ".h".

definition

A function definition defines what the function is doing. You might have noticed that there is a potentially strange keyword in the functions.c file, **void**. The return type void means that the function does not return anything.

Try to understand why the function **increment_wrong** does increment **a** inside the function scope but the value remains unchanged inside the **main** function. To your help we have written a function that does increment **a** correctly. Keep pointers in mind when you try to understand the code. If you don't understand, no worries, just know that you can't modify values inside functions. This is probably one of the more perplexing aspects of C to start with. If you want to know the reason, ask one of the teaching assistants.

This discussion about semantics on stack overflow might help you to understand (especially if you are familiar with C++).

Compile and run the program:

```
gcc -O0 functions.c -o functions && ./functions
```

Returning stack arrays

Try to run the program "stack_array". This program will run, however, it has **undefined behaviour**. Undefined behaviour is probably one of the most frustrating aspects of C. This essentially means that your code does not follow the C standard, and there are not rules for how compilers should handle it. However, the code is still compiling since you don't have any syntax errors. This means that your output will likely depend on the compiler you are using. Even worse, your code might do exactly what you expect it do, however, it does it for the wrong reason. It can therefore be hard to catch errors with undefined behaviour when the problem changes slightly (and you don't get the expected outcome anymore). The compiler can in some cases help you catch errors with undefined behaviour and these features will be discussed in the debugging exercise.

The explanation of why this is undefined behaviour is rather involved and you don't have to understand it, just know that you should not use stack pointers outside of the declared scopes. Try to fix the code by using the heap instead (see heap.c).

Compile and run the program:

```
gcc -O0 stack_array -o stack_array && ./return_stack_array
```

Uninitialized data

The program "uninitialized.c" is trying to illustrate that you can never be certain what data declared variables will hold, this is why you have to initialize them. You might have to run the program a few times to see the randomness of the program.

There is not to much you have to understand about the program. However, the take away is that you should: **never trust that your variable will have the value zero when you declare it. This is something that might expect from other languages.**

Compile and run the program:

```
gcc -O0 uninitialized -o uninitialized && ./uninitialized
```