HIGH PERFORMANCE PROGRAMMING
UPPSALA UNIVERSITY
SPRING 2020
LAB 2: MORE PROGRAMMING IN C, TIME MEASURING AND REDUCING INSTRUCTIONS

The aim of the first part of this lab (Tasks 1-7) is to continue practicing the fundamental concepts of the programming language C. Note that in Task 7, you practice how to measure time for C programs, you will need these skills in the assignments later in the course.

The aim of the second part of this lab (Tasks 8-11) is to demonstrate and explore ways of reducing the amount of work in a program.

In this lab you will write your own code or complete provided code.

There are a couple of extra tasks in the lab. Look at them if you are done with other tasks and have more time. If you need all your time for the non-extra tasks, then don't worry about the extra tasks.

Note. You are not required to write makefiles in order to compile your code.

Log into a system and download the lab tar-ball `Lab02_MoreC_Instructions.tar.gz` from the Student Portal. Save it and unpack.

Part 1. Programming in C

1. The GDB debugger

Task 1:

The code for this task is in the `Task-1` directory.

In Lab 1 we saw how GDB can be used to detect where in your code a "segmentation fault" error happens. Now we will look at some other ways of using gdb.

We will use the program `littlecode.c` as our example here. Suppose that we are interested in the state of the program when a certain function is called. We can then use gdb to set a so-called *breakpoint* in that place in the code. Then gdb will run the code until it reaches the breakpoint, and then stop, giving us the opportunity of investigating e.g. what values different variables have at that point of our program's run.

To test this, first compile the program using the `-g` compiler option to include the debugging information that gdb needs:

```
gcc -g littlecode.c -o littlecode
```

Date: January 26, 2020.

Run the program to see what it does; it computes a number x that is printed. Now run gdb giving out program as input argument, like this:

```
gdb ./littlecode
```

Now we get a gdb prompt "`(gdb)`" where we can give different commands. Suppose we are interested in the program's behavior when the `hh()` function is called. Then we can set a breakpoint there, like this:

```
(gdb) break hh
```

Then tell gdb to start our program by giving the command "run". Now gdb will run our program until it reaches the breakpoint, and then it will stop and show the "`(gdb)`" prompt again. So now we are at the breakpoint. Try the "`bt`" (backtrace) command to show the current call stack; the functions that have been called to reach this point. Compare to the code in `littlecode.c`. Does the call stack look as you expect?

When we have stopped at a breakpoint we can also check the current values of different variables using the "`print`" command. Try this, for example to check the values of the `global1` and `global2` variables. Does it work? We can use the command "`c`" (continue) to continue execution of the program. In this case, since the `hh()` function is called in a loop, the program will again stop at the same breakpoint in the next loop iteration. Try doing "`c`" several times and look at the output this gives. You can see that it stops at the breakpoint in the `hh()` function, with different values of the `a` input argument. Do the values of `a` shown match your expectations, considering how the code in `littlecode.c` is written?

Another useful thing we can do with gdb is to find out why a program seems to hang. This is a common type of bugs that can be difficult to track down – if your program just appears to freeze it is hard to know what went wrong. In this situation gdb can help.

To see how gdb can help us when our program seems to hang, start by introducing a bug in the `littlecode.c` code: change the loop condition in the loop in the `gg()` function from `c<100` to `c<200`. Since `c` is of type `char` this means that the loop will never end. Now we will pretend as if we do not know that, we just have a program with a bug that causes it to seemingly freeze while it is running.

Run the program:

```
./littlecode
```

So now the program just seems to hang, it never finishes. We can see the process ID (PID) of the program if we run `top` in another terminal window. Do that, and note the PID of the `littlecode` process. Knowing the PID we can run gdb and attach it to the process that is already running. If the PID is 1234 then this is done as follows:

```
gdb -p 1234
```

Try that. What happens?

If you get an error message saying "Could not attach to process", see the alternative approach under "Alternative" below.

When attaching gdb to a running process like this, gdb will stop that program and let us check what the program was actually doing. In this case we will directly see that the program is currently inside the `gg()` function. We can use the "`bt`" command to see which other functions were called to reach this point, and we can print values of variables that we are interested in. Hopefully this will help us understand what had happened, in this case that we have an infinite loop due to the `c<200` code change.

Alternative: On some systems, depending on security settings in Linux, gdb may not be allowed to attach to another process in the way described above. In that case, if you have a program that appears to hang and you want to stop it and find out where and why it hangs, you will have to start the program using gdb, then when it hangs enter `<ctrl>C` (where `<ctrl>` means the ctrl key on your keyboard) to interrupt the program. Then gdb will detect the interrupt and stop the program, and you will then see the gdb command prompt and be able to check what is going on in the same way as if you had attached gdb to a running process.

There is much more you can do with gdb – look at `man gdb` and/or type "help" within gdb to get more information. For example, you can look up the gdb commands "`next`" and "`step`", and try using them.

2. Multidimensional arrays

Task 2:

Write a C program to display a matrix of size n which has values -1 below the diagonal, 0 on the diagonal and 1 above the diagonal. Let your program reads the matrix size from the command line. In this task use dynamically allocated two-dimensional array for a matrix representation. First store the whole matrix as a two-dimensional array in memory, and then use the values from the array to output the matrix.

For example, the matrix of size 5 should look like:

$$\begin{matrix} 0 & 1 & 1 & 1 & 1 \\ -1 & 0 & 1 & 1 & 1 \\ -1 & -1 & 0 & 1 & 1 \\ -1 & -1 & -1 & 0 & 1 \\ -1 & -1 & -1 & -1 & 0 \end{matrix}$$

Note. Do not forget to free allocated memory at the end.

3. Pointers to functions

Short summary of how function pointers work

Example: we have a function `int myfunc(int* x, double y)`.

- Declaration of a function pointer `foo` to a function with two input parameters of types `int*` and `double` respectively, and return value of type `int`:

  ```
  int (*foo)(int*, double);
  ```

- Assigning a value to the declared pointer:

  `foo = &myfunc;` or `foo = myfunc;` (both ways are allowed)

- Invoking the function pointed to by the pointer `foo`:

  `foo(x, y);` or `(*foo)(x, y);` (both ways are allowed)

Task 3:

You are given the following two functions:

```
void print_int_1(int x) {
  printf("Here is the number: %d\n", x);
}
void print_int_2(int x) {
  printf("Wow, %d is really an impressive number!\n", x);
}
```

Write a C program which creates a function pointer to that kind of functions and calls `print_int_1` using this function pointer, then set the pointer to point to `print_int_2` instead, and use the function pointer to call that function. Does it work? Note that the same function pointer can be used to refer to different functions, as long as the return type and argument types are the same.

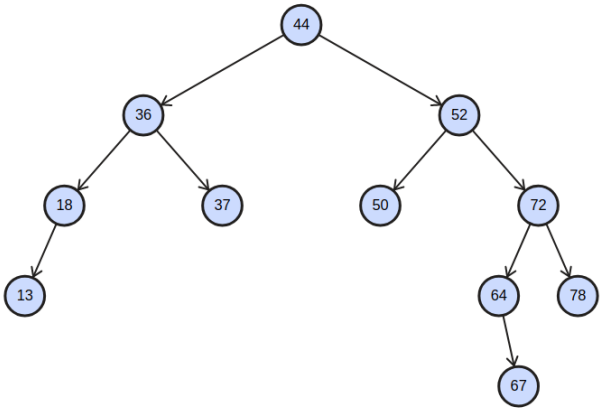
4. Binary search tree

A binary tree is a tree in which each node is allowed to have no more than two children, usually called left and right. A binary search tree (BST) is a special kind of binary tree where the data is structured using additional requirements:

- there must be no nodes with the same value
- for every node its right subtree (the tree that has its right child as root) should consist only of nodes with values greater than its value

Use this BST Visualizer for creating examples and better understanding:
http://btv.melezinek.cz/binary-search-tree.html

One example of a BST:



The code required for this section is located in the directory `Task-4`. This code will be used in tasks 4, 5 and 6. Note that you can use the GDB debugger for finding bugs in your code.

In this section we will implement a simple plant database for a botanical garden. We will use a binary search tree where each node represents a plant. Each plant has a unique ID represented by an integer number and a name represented by a string. Nodes in the binary search tree are sorted by the plant ID.

Each node is given by a structure

```
typedef struct tree_node
{
int ID;
char * name;
struct tree_node *left;
struct tree_node *right;
} node_t;
```

At the start the tree is empty:

```
node_t *root = NULL;
```

The function `print_bst` located in the file `bst_test.c` for each node prints ID, name and IDs of the left and right child nodes. For example, the output line `445 sequoia: L162,R612` means that the node with ID 445 and plant name "sequoia" has two child nodes: the left child with ID 162 and right child with ID 612. If there are no nodes inserted in the tree, you will get the message that the tree is empty. In the `bst_test.c` file functions `insert`, `delete_tree` and `search` are declared but not implemented (you will get a warning message when you run the executable). In the following tasks you will need to implement these functions.

Task 4:

Implement a function `insert` which inserts the information about the plant into the tree such that the properties of the binary search tree are preserved. If the node with a given ID already exists in the tree, do not add it in the tree, print a warning message and continue the execution.

For example insert nodes (ID, name):

- 445, sequoia
- 162, fir
- 612, baobab
- 845, spruce
- 862, rose
- 168, banana
- 225, orchid
- 582, chamomile

then the final tree should look like:

```
445 sequoia:    L162,R612
162 fir:        R168
168 banana:     R225
225 orchid:
612 baobab:     L582,R845
582 chamomile:
845 spruce:     R862
862 rose:
```

Create this tree using the BST Visualizer.

Hint. Start by adding the root node: allocate memory for the node, copy data into the node and set both child nodes to NULL. To add more nodes run the insert function recursively on the corresponding tree branch: on the right if wanted ID value is greater than the ID of the current node and on the left otherwise.

Note. Since we want the `insert` function to be able to modify the node pointer, the function has as an input parameter of type `node_t**` (a pointer to a pointer to a `node_t`).

Note. In each node memory for a structure member `name` should be allocated dynamically when the node is created. If you want, you can use the function `strdup(const char *s)` from the header file `string.h`. The function allocates sufficient memory for a copy of the string s, copies s, and returns a pointer to the copy. Note that if the string is not needed anymore, the returned pointer must be passed to `free` to avoid a memory leak. Alternatively, you can allocate memory yourself by calling `malloc` and then copy the data of the string yourself.

Task 5:

Implement a function `delete_tree` which recursively deletes all the nodes in the tree.

Hint. The empty tree is represented by a NULL pointer. After deleting the tree, we should set the `root` pointer to NULL inside the `delete_tree` function. In general, it is a good practice to set a pointer to NULL after calling function `free`. Since we want the `delete_tree` function to be able to modify the node pointer, the function has as an input parameter of type `node_t**` (a pointer to a pointer to a `node_t`).

Hint. Do not forget to free memory allocated for names.

Task 6: (Extra part)

Implement a function `search` which searches the tree to find the name of a plant with a given ID.

Example:

```
search(root, 168);
search(root, 467);
```

Output:

```
Plant with ID 168 has name banana
Plant with ID 467 does not exist!
```

5. Time measuring

Task 7:

The code for this task is in the `Task-7` directory.

In the instructions for this task we assume that you have compiled each program in such a way that the executable is named like the C source file without the `.c` extension, e.g.:

```
gcc -o regularcode regularcode.c
```

but of course you can choose to call your executable files whatever you want.

An easy way of measuring time is to use the `time` command. For example, after you compile the executable `regularcode`, run "`time ./regularcode`". After the program completes, `time` will present three timing measurements. The first one, `real` time, is the actual real time (also called wall clock time) taken. The one in the middle, `user` time, is the CPU time spent executing user code. Normally for a single-threaded program mostly doing computations the `user` time should be close to the `real` time. The third value, `sys` time, is the CPU time spent doing system calls. This should in most cases be small, but if your code is for example doing a lot of small `malloc()` and `free()` calls the `sys` time may become significant.

The `Task-7` directory contains four different codes with different behaviors. Look at each code to see what it is doing. Then compile and run each of them, using the `time` command to measure timings. The code `regularcode` is simply a serial (single-threaded) program doing some computation, so for that we expect the `user` time should be close to the `real` time. The code `sleepycode` also calls `sleep()` a few times, `mallocycode` performs lots of dynamic memory allocations, and `threadedcode` performs some computation using two threads. Check how the

different timings given by the `time` command behave for these different cases. Do they behave differently? Can you understand why?

Note: `threadedcode` uses pthreads, so you will need to link with `-lpthread` (the pthreads library) in order to build that code. You will learn more about threaded programs later in the course.

For the rest of this lab, focus mainly on the middle timing, `user` time, which is often the most informative. System "noise" e.g. disturbances due to other processes running on the same computer can greatly affect measurements — it is a good idea to run at least five times and record the *lowest* time.

3. Use the `gettimeofday()` function from `time.h` to measure the time needed to execute the for loop in `regularcode.c`, and use `printf` statement to output the timing. *Hint:* it can be convenient to write a small help function like this:

```
double get_wall_seconds(){
struct timeval tv;
gettimeofday(&tv, NULL);
double seconds = tv.tv_sec + (double)tv.tv_usec/1000000;
return seconds;
}
```

See `timings.c` and Lecture 3 slides for an example usage of `gettimeofday()`.

4. Repeat step 3, but now use `clock_gettime()` (remember to link with `-lrt`) if you're on Solaris or Linux, or `clock_get_time()` if you have a Mac. The following url contains a useful code snippet: https://gist.github.com/jbenet/1087739. See Lecture 3 slides for a `clock_gettime()` usage example.

Part 2. Reducing instructions

Many of the techniques here can be done automatically when the compiler optimizes code, but sometimes it helps to do things by hand. However, hand-optimizing can also confuse the compiler and result in *worse* performance: never optimize without a good reason and always check performance and correctness afterwards!

Throughout this lab and in future labs, compare the effects of your optimizations with and without compiler optimizations. Unless the lab instructions suggest otherwise, try at least the `-O2` and `-O3` compiler optimization flags. See Lecture 4 and/or the GCC manual for a description of different optimization flags available.

Of course, our goal should normally be to achieve the best performance we can using both compiler optimizations and manual code changes, so when making a change in our code the most important question is if that change improves performance when we have compiler optimizations turned on. However, to better understand what is going on and what the compiler is doing, it can be helpful to compare results with and without compiler optimizations.

Task 8:

The code for this task is in the `Task-8` directory.

The most basic optimization of a comparison is called the *boolean short circuit*. In C (and Java, among other languages), the evaluation of a boolean expression is terminated as soon as the outcome is known.

- `A && B`: If A is false, then the statement is false regardless of B.
- `A || B`: If A is true, then the statement is true regardless of B.

In both cases, B is not evaluated at all. This is useful for instance if B is an expensive function, but also when it's necessary to avoid causing the side-effects of evaluating B.

For example:

```
if(p != NULL && *p > 3.9) {
  /* Do something.. */
}
```

ensures that `p` is not dereferenced when it points to NULL and would cause a "segmentation fault" error.

In `short-circuit.c`, you'll find a code that repeatedly generates some random boolean values and sets a variable according to certain rules. Improve the formulation of the if-statements so they are more efficient.

Task 9:

The code for this task is in the `Task-9` directory.

In this task, we're going to cover several methods of reducing the computational load of some commonly seen tasks.

Copying arrays or initializing array values to zero is a common practice. The `string.h` library provides functions that perform these operations in one go. The program in `memset.c` shows how `memset` and `memmove` can be used. Read the code and form an expectation of the performance difference. Measure and compare the speed of the two versions. Does the actual speed difference match your expectation?

If a function is going to be called with only a limited set of inputs, consider implementing the function as a lookup table. The program in `lookup.c` shows how this can be done. Read the code and form an expectation of the performance difference. Measure and compare the speed of the two versions. Does the actual speed difference match your expectation? What sort of functions are candidates for this kind of optimization?

Note that although in the lookup table example in `lookup.c` the table was small and could be statically allocated and filled with hard-coded values, it is also possible to use larger tables that are allocated and initialized when the program runs. If the same values are computed many times during program execution, saving precomputed values in a table is often a good idea.

Task 10:

The code for this task is in the `Task-10` directory.

The computational cost of arithmetic operations varies a lot. Choosing the cheapest arithmetic formulation can improve performance significantly. This technique is known as *strength reduction*.

The program in `strength_reduction.c` contains a loop with a number of arithmetic operations. Reformulate them to improve the speed of the code (see list of hints below). Remember to check that the computed values are still the same.

The program in `math_functions.c` contains a loop calling a math function. See comment in the code for examples of other ways of getting the same result. Try out the different approaches. Which way is fastest? Can you understand why?

Here are some hints:

- The cheapest operations are integer + and -, and bit-level operations like `>>`, `&`, and `&&` .
- The bitwise shift operators `>>` and `<<` are equivalent to integer division or multiplication by a power of 2, respectively.
- Integer division by a constant is faster than with a variable.
- Integer division is faster if unsigned.
- Floating-point multiply is much faster than floating-point division.
- Arithmetic operations are faster than function calls. Don't use `pow()` if you can avoid it.
- Math functions exist in different variants for different precision, e.g. `sqrt()` and `sqrtf()`. A higher precision result is usually more expensive to compute.

Task 11: (Extra task)

Calling a function incurs a certain cost. Execution flow must jump to a different address in the code, and return. This jump alone can take up to 4 cycles and can reduce the efficiency of the instruction cache. A new stack frame is set up, parameters are stored (on the stack in 32-bit mode which takes even more time), and the registers from the previous frame must be saved and restored.

It is therefore a good idea to limit function calls in the critical part of the code. One way of doing that while maintaining code quality is with *function inlining* with the `inline` keyword.

When the compiler inlines a function, it replaces the function call with the function code, effectively removing the associated costs. Inlining is especially important for functions called from the innermost loop of a program, but can also be an effective optimization tool when used to turn a *frame function* into a *leaf function*. A frame function is a function that calls at least one other function. A leaf function is a function that doesn't call any other function. A leaf function is simpler than a frame function because the stack unwinding information can be left out if exceptions can be ruled out or if there is nothing to clean up in case of an exception. A frame function can be turned into a leaf function by inlining all the functions that it calls.

One downside of function inlining is that the compiler has to make a non-inlined copy of the inlined function, because of the possibility that another compilation unit (.c file) contains a call to the function. This is often dead code, which can impact instruction caching and executable size. Adding the `static` keyword to the function definition tells the compiler that the function can only be called from the same compilation unit. The linker option `-ffunction-sections` allows the linker to exclude unreferenced functions from the executable.

Note that the `inline` keyword does not *force* the compiler to inline the function. If the compiler decides it's a bad idea (e.g. because the function is called from too many places or is too big), then the function is not inlined. Conversely, if it determines that there is a benefit the compiler may inline functions that are not marked with the `inline` keyword.

In this extra part, construct a program (or set of programs) that demonstrates the potential benefit of function inlining. Note that you have to compile with a sufficient optimization level for inlining to occur.