EL2310 Scientific Programming
LAB2: C lab session

Patric Jensfelt, Andrzej Pronobis

Chapter 1

Introduction

1.1 Reporting errors

As any document, this document is likely to include errors and typos. Please report these to `pronobis@kth.se`.

1.2 Acknowledgements

Valuable feedback and error reports on this document have been provided by:

- T. Gezork 2008

Thank you!

1.3 Before getting started

Please see the course materials for links to tutorials about Linux/Unix if you are not familiar with this operating system. If you have access to Linux (on your own Laptop or a CSC computer), you can solve these exercises there. Otherwise, you should download the virtual machine available from the course materials page: `http://www.pronobis.pro/teaching/el2310/course-materials`.

Chapter 2

Lab instructions

2.1 Getting help

On Unix/Linux systems it is possible to get help regarding the syntax of a certain function by using the `man` command. These so called man-pages contains information about what include files to use, what arguments to pass and what the return values are.

Task

Test the man function on the functions `atoi` and `atof`. The syntax is
`man atof`

2.2 Repetition

Repeating a certain calculation or operation a number of times is a very common task. In C there are (at least) three ways to accomplish this. These are `for`, `while` and `do-while`.

Task

Write programs `loop-for.c`, `loop-while.c` and `loop-do-while.c` which generates the following output on the screen using different ways to do the iteration[1].

```
1
2
4
8
16
32
```

[1] For more information see slides from lecture 7

2.3 Command line arguments

In many cases you want to be able to control the execution of your program somehow. One way to do this is to have the user input information during the execution. Another way that is frequently used in the Unix/Linux world is to provide the program with command line arguments.

2.3.1 Syntax

To get command line argument you need to add some input parameters to your `main` function.
`int main(int argc, char *argv[])`
Here `argc` tells how many input arguments there are and `argv` contains the input arguments.

Task

Write a program, `parse_input.c` that iterates through the input arguments and prints them one by one on the screen. Remember that `char *argv[]` is an array of `char*` , i.e. character arrays or strings (C style). Try with the following

- `.\parse_input`
- `.\parse_input 1`
- `.\parse_input hello world`
- `.\parse_input -h`
- `.\parse_input -x 4`

What is `argv[0]`?

2.3.2 Converting `char*` to `int` and `double`

You often want be able to input numerical values to your program. However, all inputs from the keyboard are in the form of characters (arrays). To convert from character array to a number we can use the functions `atof` (`char*` to `double`) and `atoi` (`char*` to `int`).

Task

Write a program to test `atoi` and `atof`. Try for example the following

- `printf(``%d\n'', toi(``1''));`
- `printf(``%f\n'', tof(``1''));`
- `printf(``%d\n'', toi(``1.2''));`
- `printf(``%f\n'', tof(``1.2''));`
- `printf(``%d\n'', toi(``1Sven''));`
- `printf(``%d\n'', toi(``Sven''));`

2.3.3 Parsing of command line arguments

If you only want to pass in a single value it is easy to do by simply using the `argv`-array directly. However when you start to get many different possible command line arguments and in addition you might want to be able to put them in different order on the command line the following construction is quite useful

```
#include <unistd.h>
 ...

int main(int argc, char *argv[]) {
   const char *optstring = "i:j";
   char o = getopt(argc, argv, optstring);
   while (o != -1) {
      switch (o) {
      case 'i':
         printf(``The argument for -i is \"%s\"\n",optarg);
         break;
      case 'j':
         printf(``This option has no argument");
         break;
      case '?':
         fprintf(stderr, "Usage:  ....'');
         return -1;
      }
   o = getopt(argc, argv, optstring);
   }
...
}
```

Task

Write a program, `coords.c`, which you can send in x and y coordinates with command line options `-x` and `-y`. The variables `x` and `y` should be given default values `x=1` and `y=2` which can be overridden on the command line. The program should print out the value of x and y at the end and the sum of these. The following two calls should be possible

- `.\coords -x 3 -y 4`
- `.\coords -y 4 -x 3`
- `.\coords -x 3`
- `.\coords -y 4`

2.4 Pointers

Pointers are special variables which contain the address of a variable. Knowing the address to where the variable is stored makes it possible to change the value of the variable for example. The following code declares an integer variable and

a pointer to an integer, this pointer is then set to point to the integer and finally the value of the integer variable is changed using the pointer to it.

```
int a;
int *p;
p = &a;
p = 4;
```

Task

Write a program with one function (in addition to the `main` function). The `main` function should call the other function which return the evaluation of a function, say $cos(x)$ where x is the first argument to the function. The function should be written such that a second output containing the derivative of the function can be given by using a second pointer argument, i.e. `double eval(double x, double *dfdx);`. The function should be implemented so that the caller can choose not to have the derivative calculated. **Hint:** You can test if the address passed in is NULL for the second argument.

2.5 Pointers to functions

Just like you can define pointers to variables you can define pointers to functions. To define a variable `p` which is a pointer to a function that returns a `double` and takes a `double` as argument you would write

```
double (*p)(double);
```

Notice that `p` is the name of the function. You assign a value to the pointer by

```
p = fcn1;
```

assuming that `fcn1` is the name of a function with the same interface as above. You can then use the function by dereferencing the pointer

```
(*p)(4.2)
```

You can also have an array of function pointers.

```
double (*p[4])(double);
```

declares an array of 4 function pointers. You assign values by

```
p[0] = fcn1;
```

and you use them as

```
(*p[0])(4.2);
```

Task

Write a program that defines several different functions with the above interface, i.e. returning a `double` and taking a `double` as argument. Create an array of pointers to these functions and loop over them and call them with some argument and print the return value. The functions could be for example `x+x`, `x*x`, `sqrt(x)`, ... Use this to create a nice looking table along the lines of

```
   x  x+x   x*x  sqrt(x)
 1.1  2.2  1.21     1.05
 1.2  ...
```