EL2310 – Scientific Programming

Lecture 7: Programming in C

Ramviyas Parasuraman (ramviyas@kth.se)

Royal Institute of Technology – KTH

Overview

Lecture 7: Programming in C

- Wrap Up
- Some basics
- Strings
- Splitting code
- Makefiles

Wrap up

- Constant values: `const type <identifier> <value>`
- Preprocessor: `#define <identifier> <value>`
- Arrays: `int i[10], j[2] = {1,2}, k[] = {1,2,3}`
- Matrix: `int i[2][2] = {1,2,3,4}, k[][3] = {1,2,3}`
- Logical operators: `<=, >=, ==, !=, <, >`

- What does it do after you run the program? : `echo $?`

Wrap up

- Functions:
- Syntax:

```
return-type function-name([arguments])
{
  declarations
  statements
}
```

- Example: `#includes`

```
#defines

function declarations

main() { ...}

function definitions
```

Task 6.1

- Write a program that multiplies two matrices and prints the result

Lecture 7: Programming in C

Remember the steps

- **Write**
- **Compile**
- **Link**
- **Execute**

Linking to extra libraries

- Often use function defined in other libraries, such as `cos, sin, exp` from libm
- Need to tell linker that it should use libm as well
- `gcc -o mymathprg mymathprg.c -lm`
- Take a look at:
  `http://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html`
  `http://www.tunl.duke.edu/documents/public/root/material/5/An_Introduction_to_GCC-Brian_Gough.pdf`

enum

- enumeration constant
- An alternative to using many `#define`
- Ex:

```
enum state { STATE_START, STATE_RUN,
STATE_STOP};
```

- First name assigned value 0, next 1, etc
- The same with `#define`

```
#define STATE_START 0
#define STATE_RUN 1
#define STATE_STOP 2
```

- Can give value to all names manually
- Unassigned names will be assigned “last + 1”

Exercise

- Test enum
- What if you add as a last item `NUMBER_OF_ITEMS` in the enum?

Task 7.1

- Write function that returns the probability to draw a certain value x given that it is from a normal distribution $\mathcal{N}(\mu, \sigma)$
- `double getprob(double x, double mean, double sigma);`
- Print a table with x and p(x)

Hint: You will have to include <math.h> and link with libm (math)

Lecture 7: Programming in C

`char` array: C style strings

- Ex: `char name[] = "Toulouse";`
- `strlen(...)` return length of a string
- A string is terminated by `\0`
- The variable `name` will be of length 9 where last character has value `\0`

Hint: You have to include <string.h>

Task 7.2

- Experiment with char arrays, strlen and sizeof
- What if `char name[]= "John Smith"`, what is the string length?
- What is the array size in bytes?
- What happens if you set `name[4] = 0;`

Precedence

Incomplete table of precedence

1. `()` `[]` `->` `.`
2. `!` `~` `--` `++` `&`
3. `*` `/` `%`
4. `+` `-`
5. `>` `>=` `<` `<=`
6. `==` `!=`
7. `&&`
8. `||`
9. `=` `+=` `-=` `...`

Evaluating logical expressions

- Logical expressions are evaluated left to right
- Guaranteed to stop as soon as expression value is determined
- A logical expression that evaluates to true is assigned value 1
- A logical expression that evaluates to false is assigned value 0

Task 7.3

- Write function `double atof(char s[])`
- Should take a char array as input and return a double representation of the string
- Assume that the string is a number like `-1.234` or `123.4`

Hint: Functions `isdigit, isspace` from stdlib.h are useful
`http://www.asciitable.com/`

Lecture 7: Programming in C

Splitting code into separate files

- Can split code in a program into many files
  - Easier to read large programs
  - Makes code reuse easier
- Code is traditionally split into:
  - Header files (myunit.h) - contain mostly declarations
  - Source files (myunit.c) - contain mostly definitions

Header files

- Contain declarations of the functions defined in source files
- Are included into other files using `#include`
- The preprocessor combines all #included files into a single file before compiling
- Why do we need source files? Why not put all source code to header files?
  - Every time we make a small change in any of the #included files, the whole program has to be re-compiled
  - We clutter our files with all the definitions. For readability, it's better to split definitions and declarations

#include

- To include function declarations we use #include
- You can do
  `#include <file.h>` or
  `#include "file.h"`
- The difference is in the order in which directories are searched
- `"file.h"` version starts to look for files in local directory
- `<file.h>` looks in the included path

Splitting declarations and definitions

- Create myunit.c and myunit.h files for each code unit
- Put definitions of your functions and “private” code to .c
- Put declarations and “public” code to .h
- The header file becomes the interface of your code unit
- Files using the “public” functions of myunit.c contain:
  `#include "myunit.h"`
  to get access to declarations and be able to use the unit.
- myunit.c should also include myunit.h
- Compile with `gcc -o program main.c myunit.c`
- If you change something in myunit.c only myunit.c will be re-compiled

Avoiding multiple definitions

- Each variable/function can only be defined once
- What if you include a file that includes a file, that includes a file, etc
- File can be included twice - we might get multiple definitions

Avoiding multiple definitions

- To avoid multiple declarations use “include guard”:

```
#ifndef __MYUNIT_H__
#define __MYUNIT_H__

double function1(double x);
double function2(double x, double y);

#endif
```

  in the header file
- Make sure that the symbol, here `__MYUNIT_H__` is unique

Task 7.4

- Implement a Newton to $f(x) = cos(x) - x^3$

$$x_{n+1} = x_n - \frac{f(x)}{f'(x)}$$

- Put the functions that evaluate $f(x)$ and $f'(x)$ into a separate file
- Convert the example Matlab code on the course page to C.

Lecture 7: Programming in C

Building projects with many files

- Method 1: Build everything in one line

```
gcc -o program program.c file1.c file2.c -lm
```

- Method 2: Compile first, then link

```
gcc -o file1.o -c file1.c
gcc -o file2.o -c file2.c
gcc -o program program.c file1.o file2.o -lm
```

The *make* tool

- When you have many files and larger projects it helps to have a tool when you compile and link your code
- *make* is such a tool
- File **Makefile** contains instructions/rules describing how to build stuff

Makefile

- `VARNAME=` declares variable
- `$(VARNAME)` access variable
- `rulename:` defines *rule*
  - `make rulename` Makes rule `rulename`
  - `make` Makes first rule
- `#` starts a comment

Standard variable names

`CC` = C compiler
`CXX` = C++ compiler
`LDLIBS` = external libraries Ex: `-lm`
`INCLUDES` = path for external declarations Ex: `-I`
`CFLAGS` = flags for the C compiler Ex: `-Wall`
`CXXFLAGS` = flags for the C++ compiler Ex: `-Wall`
`LDFLAGS` = flags for the linker Ex: `-L`

- If you do not provide a rule, one might be generated for you
- It will use those variables

Rules

- Compiles executable

```
TASK1=task1
TASK1_OBJS=task1.c functions.c
$(TASK1):
    $(CC) -o $(TASK1) $(TASK1_OBJS) $(LDLIBS)
```

- Remove created files

```
clean:
rm -f *.o $(TASK1)
```

- It is possible to specify dependencies

```
all:  $(TASK1) task3
```

- Also take a look at: `http://www.cprogramming.com/tutorial/makefiles.html`