EL2310 – Scientific Programming

Lecture 9: Scope and Pointers

Andrzej Pronobis
(pronobis@kth.se)

Royal Institute of Technology – KTH

Overview

Lecture 9: Scope and Pointers

- Wrap Up
- Splitting code
- Makefiles
- Scopes
- Pointer Basics
- Pointers and Arrays

Last time

- Arrays
- Functions
- Logical expressions
- Precedence

Today

- Splitting into separate files
- A first look at a Makefile
- Scope rules
- Pointers

Functions

- Syntax:

  ```
  return-type function-name([arguments])
  {
    declarations
    statements
  }
  ```
- If the function does return anything you give it return-type `void`
- If you return something you leave the function with statement:

  ```
  return value;
  ```

  where `value` is of the return-type
- If the function has return-type `void` you leave with `return` if you want to leave before the function ends, otherwise you do not have to give an explicit `return`
- NOTE: If your function has a return type and you do not have

Declaring functions

- A function just like a variable need to be declared before it is used
- Either put the definition of the function before it is used or,
- add a declaration of it first and then later define it
- File example:

```
#includes
#defines

function declarations

main() { ...}

function definitions
```

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
- Ex: `gcc -o mymathprg mymathprg.c -lm`

Lecture 9: Scope and Pointers

- Wrap Up
- Splitting code
- Makefiles
- Scopes
- Pointer Basics
- Pointers and Arrays

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 include the 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 1

- 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

Lecture 9: Scope and Pointers

Wrap Up

Splitting code

Makefiles

Scopes

Pointer Basics

Pointers and Arrays

Building project with many files

- Method 1: Build everything on 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 project 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
- A Makefile skeleton is provided with today's tasks

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
  ```

Task 2

Write a Makefile for Task 1

- Run make multiple times.
- What happens when you run make withouth changing the file?
- Make knows what needs to be re-compiled!

Lecture 9: Scope and Pointers

- Wrap Up
- Splitting code
- Makefiles
- Scopes
- Pointer Basics
- Pointers and Arrays

Variable scope: local variables

- The scope of a variable tells where this variable can be used
- Local variables in a function can only be used in that function
- They are automatically created when the function is called and disappear when the function exits
- Local variables are initialized during each function call

Variable scope: `extern`

- If you want to use a variable defined externally to a function in some other file, you need to use the keyword `extern`
- `extern int value;` declares a variable `value` defined externally that will now available to us

Variable scope: `static`

- If you want a variable defined outside a function to be hidden in a file, use the keyword
  `static`
- A variable declared `static` can be used as any other variable in that file but will not be seen from outside

Initialization

- External and static variables are guaranteed to be 0 if not explicitly initialized
- Local variables are NOT initialized (contain whatever is in the memory)

Task 3

- Write program with two functions: fcn1 and fcn2
- Let each function
  1. define a variable, but not initialize
  2. print the value
  3. set the value (different for fcn1 and fcn2)
  4. print it again
- Call fcn1, fcn1, fcn2 and fcn1 and see what you get
- Lesson: Initializing your variables is important!!

Lecture 9: Scope and Pointers

- Wrap Up
- Splitting code
- Makefiles
- Scopes
- **Pointer Basics**
- Pointers and Arrays

Pointers

- Pointers are special kinds of variables
- They contain the address of another variable
- Pointers are like bookmarks
- Used heavily in C:
  - To pass reference to big things in memory
  - To return multiple values from functions
- Have to be used with care

Declaring a pointer

- A pointer is declared by a * as prefix to the variable
  Can think of it as a suffix to the data type as well
  "`int*` is a pointer to an `int`"
- Ex: Pointer to an interger

  ```
  int *ptr;
  ```

Assigning a pointer

- You assign a pointer to a value being an address of a memory location
- The address typically correspond to a variable in memory
- You get the address of a variable with the unary & operator
- Ex:

```
int a;
int *b = &a;
```

- We say that b “points” to a

Dereferencing a pointer

- To get the value in the address pointed to by a pointer, use the operator dereferencing operator *
- Ex:

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

- Will set `a` to be 4

Copying pointers

- Copying the data

```
*ptr1 = *ptr2;
```

- Copying the pointer address

```
ptr1 = ptr2;
```

Passing values by reference

- Can use pointer to pass something to a function
  Ex `void func(double x, double *f);`
- The pointer is a local variable inside function, but it points to something outside the function
- Allows the function to change the variable outside
- A way to return “multiple outputs from a function”

Task 4

- Rewrite the Newton code using a function on the form instead of f1x and df1dx

```
void eval_fcn(double x, double *f, double *dfdx);
```

Pointers and arrays

- Can use pointer to perform operations on arrays
- Ex:

```
int a[] = {1,2,3,4,5,6,7,8};
int *p = &a[0];
```

- Will create a pointer that points to the first element of a

Stepping forward backward with pointers

- A pointer points to the address of a variable of the given data type
- If you say `ptr = ptr + 1;` you step to the next variable in memory assuming that they are all lined up next to each other
- Can also use shorthand `ptr++` and `ptr--` as well as `ptr+=2;` and `ptr-=3;`
- Remember `sizeof`?

Task 5

- Allocate an array and use a pointer to loop through it

Arrays and pointers

- Pointers and arrays are very similar
- Assume

  ```
  int a[10];
  int *p;
  ```
- The following are equivalent

  `p = &a[0]` and `p = a;`

  `a[i]` and `*(a+i)`

  `&a[i]` and `a+i`

  `*(p+i)` and `p[i]`

  `fcn(int *a)` and `fcn(int a[])`

More on pointers

- One has to be careful when moving pointers
- Common mistake when using a pointer: you move it outside the memory space you intended and change unexpected things
- The following is allowed but make it hard to read

```
int a[] = {6,5,4,3,2,1};
int *p = &a[2];
p[-2] = 2;
```

- What value will change?

Constant strings

- The “Hello world” in `printf("Hello world");` is a constant string literal
- It cannot be changed
- Consider the two expressions

  ```
  char amsg[] = "Hello world";
  char *pmsg = "Hello world";
  ```
- `amsg` is a character array initialized to “Hello world”. You can modify the content of the array since it contains a copy of the string literal.
- `pmsg` is a pointer that points to a constant string directly. You cannot change the character in the string but change what pmsg points to.

Task 6

- Write the function
  `void strcpy2(char *dest, char *src);`
- Should copy the string `src` into `dest`

Next Time

- Continue with pointers