EL2310 – Scientific Programming

Lecture 9: Scope and Pointers

Carl Henrik Ek
(chek@csc.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

- Functions provide a way to encapsulate a piece of code
- Gives it a well defined input and output
- Makes code easier to read
- (Often do not have to read code in the function)

Functions, cont'd

- Syntax:

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

- If the function does return anything you give it return-type `void`
- If you return something you leave the function with a statement like

  `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`

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

Splitting code into separate files

- Can split code in a program into many files
  - Easier to read large programs
  - Makes code reuse easier
- Ex: main.c and myfunctions.c
- Compile with `gcc -o program main.c myfunctions.c`

Declaring and definition

- If you have separate files you need to make sure that the functions are declared before you use them
- If you move functions into separate files you can create a *header* file (h-file) where you declare the functions in the corresponding source file (c-file)
- Ex: The file myfunctions.c would be accompanied with a file myfunctions.h where the functions in myfunction.c are declared
- Files using myfunctions.c then #include “myfunctions.h” to get these 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

Avoiding multiple definitions

- Each variable/function must be declared before used
- Each variable/function can only be defined once
- What if you include a file that includes a file, that includes a file, etc

Avoiding multiple declarations

- To avoid multiple declarations use construction like

```
#ifndef __MYFUNCTIONS_H__
#define __MYFUNCTIONS_H__

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

#endif
```

  in the header file
- Make sure that the symbol, here `__MYFUNCTIONS_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
- The function should have parameters for max number of iterations and precision

Lecture 9: Scope and Pointers

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

Makefile

- = declares variable
- `$` access variable
- `:` defines *rule*
  - `Make <foo>` Makes rule `<foo>`
  - `Make` Makes first rule
- provided “skeleton” todays task

Variables

`CXX` = compiler id
`LIBS` = external libraries Ex: `-lm`
`INCLUDES` = path for external declarations Ex: `-I`
`CXXFLAGS` = flags for the compiler Ex: `-Wall`
`GETSCANS` = executable name
`GETSCANS_OBJS` = source

Rules

- Compiles executable

```
$(GETSCANS):
$(CXX) -o $(GETSCANS) $(GETSCANS_OBJS)
$(INCLUDES) $(LIBS)
```

- Remove created files

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

Task 2

Write a Makefile for Task 1

Lecture 9: Scope and Pointers

Variable scope: automatic 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
- These variables are also known as *automatic*
- They are automatically created when the funcion is called and disappears when the function is exited
- Automatic variables need to be initialized on each function call
- Will contain garbage otherwise

Variable scope: `extern`

- If you want to use a variable defined in some other file you need to use the keyword
  `extern`
- `extern int value;` declares a variable `value` and we let the compiler know that it is is defined somewhere else

Variable scope: `static`

- If you want a variable 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 guarenteed to be 0 if not explicitly initialized
- Automatic variables are undefined (whatevery is in the memory)

Task 3

- Write program with two functions, fcn1 and fcn2
- Let each function
  1. define a variable
  2. print its 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

Pointers

- Pointers are special kinds of variables
- They contain the address of another variable
- Used heavily in C
- Have to be used with care
- Used in the wrong way, makes programs hard to understand
- Used in the right way, makes it easier to write programs

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 the address to 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;
  ```

Pointer for function call by reference

- Can use pointer to function calls by reference
- Allows the function to change a variable
  - “Multiple outputs from a function”

Task 4

- Rewrite the Newton code using
- a function on the form

```
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

- You need to keep track of how you can move you pointers
- Common mistake when using pointer: You go outside of the 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
- 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
- `pmsg` is a pointer that currently points to a constant string. 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

- Lecture Tomorrow 10-12 M36
- Continue with pointers