EL2310 – Scientific Programming

Lecture 8: Scopes and Pointers in C

Ramviyas Parasuraman (ramviyas@kth.se)

Royal Institute of Technology – KTH

Overview

Lecture 8: Scope and Pointers

- Wrap Up
- Scopes
- Pointer Basics
- Pointers and Arrays

Wrap up

- Linking to extra libraries
  ex: for lib math: `gcc -o mymathprg mymathprg.c -lm`
- Strings: `char name[] = "Example";`
  `strlen, #include <string.h>`
- Splitting code:
  - Header files - all includes, defines, declarations, etc.
  - Source files - all definitions, main function, private code, etc.

Makefiles

- Compiles executable

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

Announcements

- C project will be announced today
- Deadline will be: 29th Sep (No excuses!)
- All course lectures will be in Bilda before lecture starts

Lecture 8: Scope and Pointers

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 be available outside the file

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 garbage values)

Task 8.1

- 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 8: Scope and Pointers

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 integer

```
int *ptr;
```

Assigning a pointer

- You assign a pointer to a value being an address of a memory location
- The address typically corresponds 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
- What's the difference between `int* and int *`?

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 8.2

- Rewrite the Newton code using a function of the following 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`?

Exercise

- 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 8.3

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

Pointers to pointers

- Can have pointers to pointer
- “Address of the address to the value”
- Notation similar
- ```
  int a;
  int *p = &a;
  int **pp = &p;
  ```
- Example use: Change address of pointer in function
- Dereferencing:
  - `*pp` to get pointer to `a`
  - `**pp` to get value of `a`
  ```

Arrays of pointers

- Can also make arrays of pointers like any other data type
- Ex: `char *sa[100];` array of 100 C strings
- Ex: `int *ia[100];` array of 100 `int` pointers

`void` pointer

- Normal pointers point to a certain type like `int`
- The `void` pointer (`void*`) represents a general pointer that can point to anything
- You can assign to and from a `void *` without a problem
- You can not dereference a `void*`
- The `void` pointer allows you to write code that can work with addresses to any data type

`void` pointer cont'd

- NOT ALLOWED

```
int a = 4;
void *b = &a;
*b = 2;
```

- ALLOWED

```
int a = 4;
void *b = &a;
int *c = b; *c = 2;
```

NULL

- Bad idea to leave variables unitialized
- This is true for pointers as well
- To mark that a pointer is not assigned and give it a well defined value we use the `NULL` pointer.
- Ex:

```
int *p = NULL;

...

if (p != NULL) *p = 4;
```

- Testing if not `NULL` before using a pointer is good practice (and setting it to `NULL` when unassigned)

Next Time

- Continue with pointers, struct