CS 240: Programming in C

Lecture 21: Callbacks, Efficiency, Libraries

PURDUE UNIVERSITY

Announcements

- Homework 9 due tonight
- Midterm 2 grades hopefully posted on Monday
- My office hours are cancelled today
    - TAs will still hold office hours

The void type

- There is a type in C that represents nothing
- It is used in only two cases:
    - To represent a function that has no return value

```
void no_value(int x) {
  printf("Value is %d\n", x);
  return;
}
```

- A pointer to something **opaque**:

```
void *pointer = NULL;
int *i_ptr = NULL;
int *i_arr = malloc(sizeof(int) * 15);
pointer = i_arr;
i_ptr = (int *) pointer;
```

What you can do to a void *

- You can assign any pointer type to a void * variable without a cast
- A void * type will hold (almost) any other first-class data type (e.g., double, int, long)
- You can later assign the void * type to a usable type again with a cast
- You may not dereference a void * type
- You should not perform pointer arithmetic on a void * type

*When to use void **

- Use the void * type to server as a conveyor of opaque data or data whose type is not yet known
- Example: the free() function:

```
void free(void *ptr);
```

  - free() does not care what type of pointer we pass it. It only needs to know where it points to.
  - This allows you to free any type of pointer

Another application: callbacks

- Suppose I set up some kind of function that accepted a pointer to a function and a value to pass to that function:

```
void setup_cb(void (*callback)(int),
              int callback_value) {
  callback(callback_value);
}
```

- This function allows the user to pass a function to call and the integer value to call it with
    - What if we wanted to use more than integers?

Generalize callback arguments

- Change the functions to use void * instead

```
void setup_cb(void (*callback)(void *),
              void *callback_value) {
  callback(callback_value);
}
```

- Now we can pass various pointer types in addition to integers and other first-class types

Callback example...

```
#include <signal.h>
#include <sys/time.h>

void *callback_data;
void (*callback)(void *);

void signal_handler(int x) {
  callback(callback_data);
}
void setup_timer(int rate, void (*cb)(void *),
                 void *cb_data) {
  struct itimerval i = { {rate, 0}, {rate, 0} };
  callback = cb;
  callback_data = cb_data;
  setitimer(ITIMER_REAL, &i, NULL);
  signal(SIGALRM, signal_handler);
}
```

And how to use it...

```
void print_msg(void *arg) {
  char *msg = (char *) arg;
  printf("%s\n", msg);
}

int main() {
  setup_timer(1, print_msg, "Sample message");
  while(1);
}
```

Another callback example

- In this example, we set up a “clock” structure and then use an asynchronous callback mechanism to update it:

```
struct clock {
  volatile char hours;
  volatile char minutes;
  volatile char seconds;
};
```

- Then we define a routine used to update it…

Another callback example

```
void update_clock(void *v_ptr) {
  struct clock *c_ptr = (struct clock *) v_ptr;
  c_ptr->seconds++;
  if (c_ptr->seconds == 60) {
    c_ptr->seconds = 0;
    c_ptr->minutes++;
    if (c_ptr->minutes == 60) {
      c_ptr->minutes = 0;
      c_ptr->hours++;
      if (c_ptr->hours == 13) {
        c_ptr->hours = 1;
      }
    }
  }
}
```

Another callback example

- Now we have a main() function that sets everything up and demonstrates it…

```
int main() {
  struct clock *clk = NULL;
  clk = calloc(1, sizeof(struct clock));
  setup_timer(1, update_clock, clk);
  while(1) {
    printf(“Hit return!”);
    getchar();
    printf(“Time: %02d:%02d:%02d\n”,
      clk->hours, clk->minutes, clk->seconds);
  }
}
```

Efficiency Issues

- Efficiency of memory vs. runtime
- Memory is not usually an issue with GiB of RAM in today’s computers, but proper use of data and its structure can play a big part in runtime
- Many methods
  - Compiler efficiencies
  - Coding efficiencies
  - Data access efficiencies

Compiler efficiencies

- gcc has optimization flags for compiling: -O<x>
  - -O, -O1: tries to “register” variables, compares multiple lines for optimization
  - -O2: optimize more without generating longer code
  - -O3: function inlining, loop unrolling, etc.
  - -Ofast: optimize for speed, disregard standard compliance
- Note: debugging tools may not work correctly with any code compiled with any optimization

Coding efficiencies

- Use local variables if the data is used more than twice in the function
- Use macros instead of short functions
- Use register variables
- Calculate what you can either before or after a loop

```
/* ok */
for (int i = 0; i < 100; i++) {
  j = i * 4.0 / bottom;
  printf(“%d\n”, j);
}
```

```
/* better */
double mult = 4.0 / bottom;
for (int i = 0; i < 100; i++) {
  j = i * mult;
  printf(“%d\n”, j);
}
```

PURDUE UNIVERSITY

Data access efficiencies

- Reuse allocated memory
  - malloc()/calloc()/free() are SLOW!

Libraries

- Remember when we had to use the `-lm` flag when using mathematical functions?
- When you use the `-lm` flag, this tells the linker to pull in the math library
    - Object code that is selectively linked in as needed

What -lm really means

- Every C development environment allows you to specify libraries
  - With gcc, you use the `-l<library>` flag one or more times
- The <library> part gets expanded into a library file named **lib<library>.so** or **lib<library>.a**, which is located on the system somewhere
- For example, using the flags `-lm` and `-lcrypto` would link in the libraries:
  - /usr/lib/libm.so
  - /usr/lib/libcrypto.so

Two types of libraries

- Static libraries
  - Become part of the executable
  - .a filename extension
- Shared object (dynamic) libraries
  - Loaded on startup and runtime
  - .so filename extension

Static libraries

- Collection of object files whose internal symbols are indexed for fast lookup by the linker
- When linking, libraries are searched for symbols that are not yet defined
- If a missing symbol is found, the object that contains the symbol is pulled into the executable
- Process is repeated until all symbols are resolved and defined

Example

- file1.c

```
float plus(float x, float y) {
  return x + y;
}

float mult(float x, float y) {
  return x * y;
}
```

Example

- file2.c

```c
/* prototypes */
float plus(float, float);
float mult(float, float);

float sub(float x, float y) {
  return plus(x, -y);
}

float div(float x, float y) {
  return mult(x, 1 / y);
}
```

Example

- Compile the two files into objects like this:

```
$ gcc -Wall -Werror -c file1.c
$ gcc -Wall -Werror -c file2.c
```

- Build a library out of the two files like this:

```
$ ar -crvs libmy_math.a file1.o file2.o
```

Now compile this with main()

- main.c

```
#include <stdio.h>
float plus(float, float); /* prototype */
int main() {
  printf("5 + 6 = %f\n", plus(5, 6));
  return 0;
}
```

- Compile and link:

```
gcc -o exe main.c -Wall -Werror -L. -lmy_math
```

  - What object(s) get pulled into the executable?

PURDUE UNIVERSITY

Now compile this with main()

- main.c

```
#include <stdio.h>
float plus(float, float); /* prototype */
int main() {
  printf("5 + 6 = %f\n", plus(5, 6));
  return 0;
}
```

-L<dir> means search in <dir> before looking in /usr/lib for the libraries

- Compile and link:

```
gcc -o exe main.c -Wall -Werror -L. -lmy_math
```

  - What object(s) get pulled into the executable?

Dynamic libraries

- Compile the two files into objects like this:

```
$ gcc -Wall -Werror -c -fPIC file1.c
$ gcc -Wall -Werror -c -fPIC file2.c
```

- Build a library out of the two files like this:

```
$ gcc file1.o file2.o -shared -o libmy_math.so
```

Same compile/link

- main.c

```
#include <stdio.h>
float plus(float, float); /* prototype */
int main() {
  printf("5 + 6 = %f\n", plus(5, 6));
  return 0;
}
```

- Compile and link:

```
gcc -o exe main.c -Wall -Werror -L. -lmy_math
```

  - What object(s) get pulled into the executable?

Why use libraries?

- The C language has no built-in functions
- You are always using a library: The C Standard Library (/usr/lib/libc.so) that contains functions like printf(), strcpy(), and similar friends
- Create your own libraries when you have a lot of object files that you need to keep organized or need to share with someone else
- Linking in a single library that contains 7,000 object files is faster than linking against 7,000 separate object files…

Example project

- Suppose I have a large software project that has the following data structures:
  - country
  - state
  - county
  - township
  - road
- There are various interactions, e.g., a county contains a list of townships, a road may contain a list of townships it connects, etc.

Rule 1: declare one data structure per file

- I might have a header file called county.h that declares a struct county:

```
struct county {
  struct township *township_array[];
  ...
};
```

- What to do about that struct township?

Two ways to handle forward references

- If a data structure is referred to only by pointer (e.g., struct township * within county), you can create a forward declaration for it:

```
struct township;

struct county {
  struct township *township_array[];
};
```

- Otherwise, you need to #include the full definition…

PURDUE UNIVERSITY

Rule 2: Use #includes in your header files

- The other way to handle townships within a county:

```
#include “township.h”

struct county {
  struct township *township_array[];
};
```

- And you can guess what’s in township.h

Rule 3: Use only as many #includes as you need

- Within county.h, we might #include lots of other stuff that is unnecessary:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <blahblahblah.h>

#include “township.h”
struct county {
  struct township *township_array[];
};
```

- Put these extra #includes in C files only

PURDUE UNIVERSITY

Rule 4: Make sure you #include a file only once

- What happens now if, in a C file, I say:

```
#include “township.h”
#include “county.h”
```

- This will create a “duplicate declaration” error
- We can use a simple and very common C pre-processor trick to avoid this

Header guards

- Add these definitions to each header file:
- township.h:

```
#ifndef __township_h__
#define __township_h__

struct township {
  ...
};

#endif  /* __township_h__ */
```

PURDUE UNIVERSITY

Avoiding duplicate #includes

- Over in county.h:

```
#ifndef __county_h__
#define __county_h__

#include “township.h”

struct county {
  struct township *township_array[];
  ...
};

#endif  /* __county_h__ */
```

Avoiding duplicate #includes

- Over in county.h:

```
#ifndef __county_h__
#define __county_h__

#include “township.h”

struct county {
  struct township *township_array[];
  ...
};

#endif  /* __county_h__ */
```

If township.h was already #included, the #ifndef will make this #include benign

Avoiding duplicate #includes

- So, back in our .c file:

#defines __township__h_

```
#include “township.h”
#include “county.h”
```

township.h contents not re-included this time!

Slides

- Slides are heavily based on Prof. Turkstra’s material from previous semesters.