CS 240: Programming in C

Lecture 21: Callbacks Wrap-up Libraries Large-scale Development

Prof. Jeff Turkstra

Announcements

- Midterm 2 Exam Thursday!
- We DO have lecture on Wednesday

Midterm 2

- May contain questions about void/void *

Feasting with Faculty

- Canceled for this week, will resume next week!
- Also no office hours!

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 using void *

- 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

A generic mechanism to run something periodically…

```
#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 something to use it…

- Now we have a main() function that demonstrates 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);
}
```

Full example of a callback

- 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…

update_clock()

```
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;
      }
    }
  }
}
```

And something to use it…

- 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 not usually an issue with GiB 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 efficiency

- gcc has optimization flags for compiling: -Ox (the letter O and number 1-3)
  - -O, -O1: tries to “register” variables, compares multiple lines for optimization
  - -O2: optimize more without generating longer code
  - -O3: function inlining, loop unrolling, etc
- Note: debugging tools may not work correctly with any code compiled with any optimization
- Change in Makefile: CFLAGS = -O2 … (line 12

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 (i = 0; i < 100; i++)
{
  j = i * 4.0 / bottom;
  printf("%d\n", j);
}
```

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

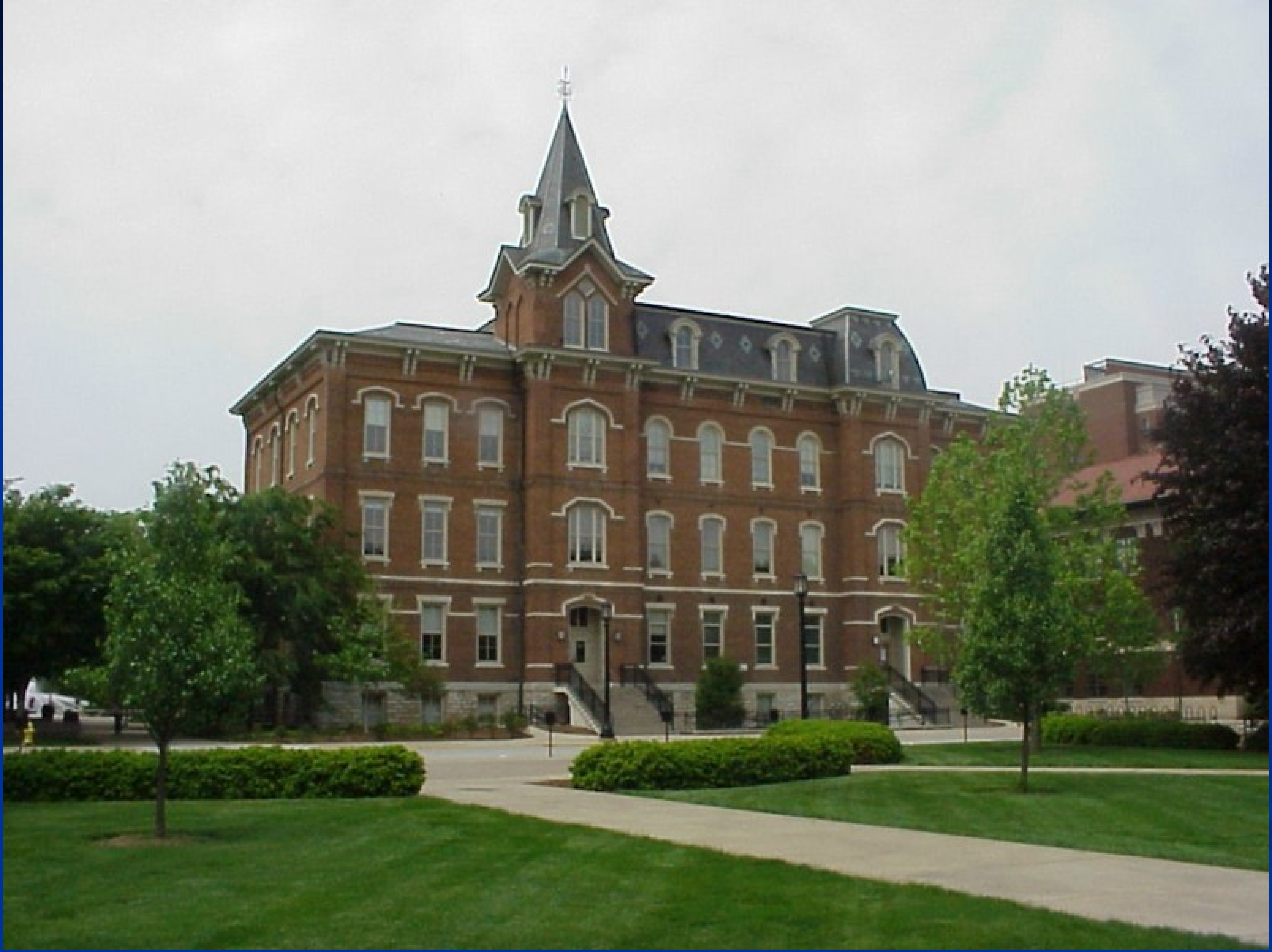
Data access efficiencies

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

Purdue Trivia

- University Hall (UNIV) is the only remaining of the original six-building campus
- Construction started Fall 1874
  - $35,000 to complete
  - Dedicated November 1877
- Used as a classroom and University's first library
- Remodeled in 1961

”John Purdue requested that he be buried in front of University Hall, and his grave directly east of the building still serves as a monument to him and the university he loved” - Mortar Board 1984



Libraries

- Remember when we had to use the -lm flag when using mathematical functions?
  - It was in the Makefile
- 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, which is located on the system somewhere
- For example, using the flags -lm and -lcrypto would link in the libraries /usr/lib/libm.so and /usr/lib/libcrypto.so

Two types of libraries

- Static libraries
  - Become part of the executable
- Shared object (dynamic) libraries
  - Loaded on startup and runtime

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 (continued)

- file2.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 (continued)

- 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: (UNIX specific)

```
ar -crv 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?

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

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: (UNIX specific)

```
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;
}
```

-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?

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 that 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 do we 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...

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.

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"
  ```
  Also #includes "township.h"
- This will create a "duplicate declaration" error
- We can use a simple and very common C pre-processor trick to avoid this

In every header file…

- township.h:

```
#ifndef __township_h__
#define __township_h__

struct township {
  …
};

#endif  /* __township_h__ */
```

- You choose the style for the symbol that you use

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 #ifdef will make this #include benign.

Avoiding duplicate #includes

- So, back in our .c file:

```
#include "township.h"
#include "county.h"
```

#defines __township_h__

township.h contents not re-included this time!

Boiler Up!