PURDUE
UNIVERSITY®

CS 240: Programming in C

Lecture 19: The C Preprocessor Casts, Void, and Callbacks

Prof. Jeff Turkstra

Announcements

- Midterm Exam 2 Thursday, April 10!
  - Sample exams and questions on the website
  - Check the seating charts!
  - 8:00pm – 10:00pm
- Feasting with Faculty this Thursday!

The preprocessor

- When a .c file is compiled, it is first scanned and modified by a preprocessor before being handed to the real compiler
- If the preprocessor finds a line that begins with a #, it hides it from the compiler and makes special note of it
  - Or, perhaps, takes other actions
- We’ve seen only two preprocessor directives so far:
  - #define and #include

#include

- #include always pulls a header file into another file
  - #include “file.h”
    Pull in file.h from the present directory
  - #include <file.h>
    Pull in /usr/include/file.h

Example of #include

/home/jeff/x.c

```
#include <stdio.h>
#include "x.h"

int main() {
  printf ("Val %d\n", X);
  return 0;
}
```

/usr/include/stdio.h

```
/*
 * scary things
 * in this file...
 */
typedef FILE ...
```

/home/jeff/x.h

```
#define X  ( 3456 )
```

Final result of #include

```
/*
 * scary things
 * in this file...
 */
typedef FILE ...

#define X  ( 3456 )

int main()
{
  printf ("Val %d\n", X);
  return 0;
}
```

- All of the things that previously resided in separate files were pulled together into one stream
- This gets fed to the compiler

More preprocessor directives

- This might be best done by example:

```
#define TESTING

  x = some_function(y);
#ifdef TESTING
  printf("Debug point!\n");
  x = x + 5;
#else
  x = x + 5;
#endif
```

- If we turn off the TESTING variable, the debug statements are no longer delivered to the compiler

More preprocessor directives

- This might be best done by example:

```
/* #define TESTING */

  x = some_function(y);
#ifdef TESTING
  printf("Debug point!\n");
  x = x + 5;
#else
  x = x + 5;
#endif
```

- If we turn off the TESTING variable, the debug statements are no longer delivered to the compiler

More preprocessor directives

- More flexible directives…

```
#if defined(TESTING) && !
defined(FAST)
  printf("Debug point!\n");
#endif
```

- You can have mathematical expressions also…

```
#define FLAG 46
#if (FLAG % 4 == 0) || (FLAG == 13)
...
#endif
```

You can #define macros…

- You can create something that looks like a function but just gets substituted at compile time:
```
#define INC(x)  x + 1
```

Notice, no semi-colon!

- So the following statement:
```
printf("I like the number %d\n", INC(z));
```
becomes, at compile time:
```
printf("I like the number %d\n", z + 1);
```

purdue university

Ternary operator

- Some C operators take one operand: &, *, -, ...
- Many C operators take two operands: +, /, %, ...
- One C operator takes three operands: `x = a ? b : c;`
  - This is the ternary operator. It means "if a is non-zero, then use the value b. Else, use the value c."
  - We typically use it in macros

More macros

- Find the absolute value:
  ```
  #define ABS(x)    x < 0 ? -x : x
  ```
- Find the highest number:
  ```
  #define MAX(x, y) x > y ? x : y
  ```
- Problems result if you say something like:
  ```
  A = ABS(B + C);
  A = B + C < 0 ? - B + C : B + C;
  ```
- So we add parentheses around the substitution variables to make them safe.

Safer macros

- Find the absolute value:

```
#define ABS(x) ( (x) < 0 ? -(x) : (x) )
```

- Find the highest number:

```
#define MAX(x, y)  ( (x) > (y) ? (x) : (y) )
```

- A longer one:

```
#define RET_ON_ERROR(x) \
    if ((x) < OK) { \
      printf(“ERROR: %d\n”, (x)); \
      return (x); } \
```

Why macros?

- Runtime efficiency
  - The preprocessor replaces the macro identifier with the token string.
  - No overhead of a function call.
- Passed arguments can be of any type. Why is this fact so cool??

  ```
  #define MAX(x, y) ((x) > (y) ? (x) : (y))
  ```

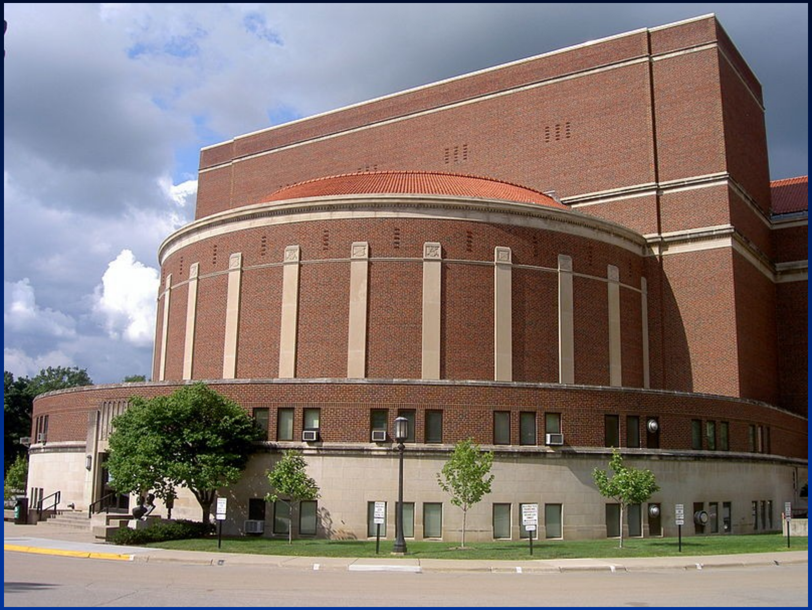
  - We only need one macro for finding the highest number regardless if the arguments were ints, floats, doubles, even chars. They all work.
    - A function called max() would not be this flexible

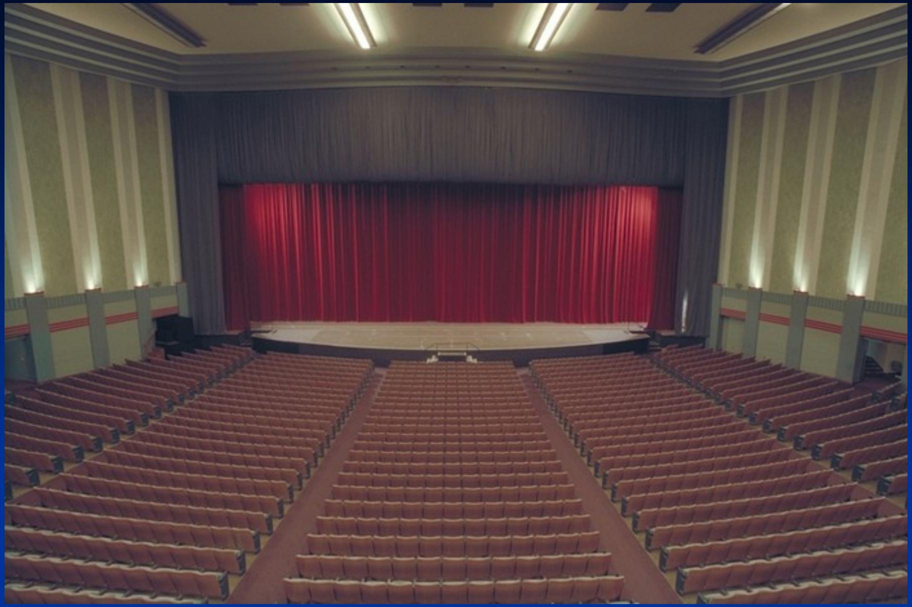
Other preprocessor tricks

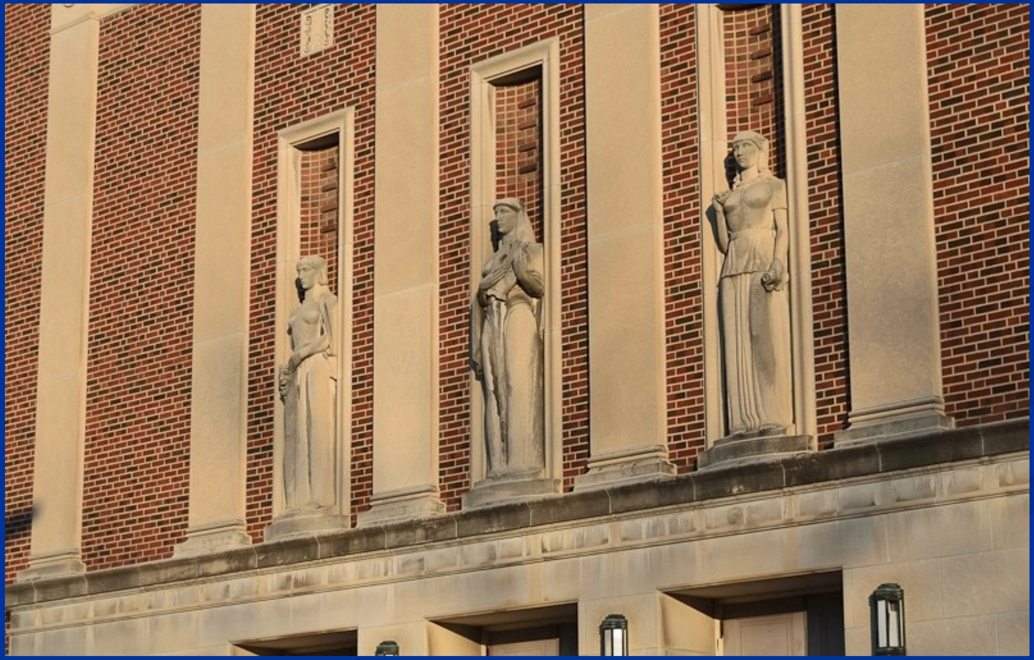
- You could spend a lot of time looking at nuances of the preprocessor
- Consider the following:
  `printf(“The date is %s\n”, __DATE__);`
- Most of the preprocessor features are for advanced software development practices.
  - If you create a large software project in C, someone in your development team should be a preprocessor expert
- Read Chapter 4.11 for more info

Purdue Trivia

- Elliott Hall of Music was dedicated on May 3 and 4, 1940 with more than 11,000 people attending
  - Included a recital by opera stars Helen Jepson and Nino Martini
- Seats 6,005 on three levels – one of the largest proscenium theaters in the world
- Named after Edward C. Elliott, president of Purdue 1922-1945
- “Sister” to Radio City Music Hall
  - J. Andre Fouilhoux, designer of New York’s Radio City Music Hall, served as one of Elliott Hall’s architects along with Walter Scholer
  - 5 seats larger
- John Johnson, an artist from Frankfort, IN created the sculptures







Throwing away type safety

- Normally, the compiler makes sure that you do not make an assignment from one type of variable to another of an incompatible type
- Disallowed example:

```
char *c_ptr = NULL;
int *i_ptr = NULL;
int *i_arr = malloc(sizeof(int) * 4);
c_ptr = i_arr;
i_ptr = c_ptr;
i_ptr[1] = 7;
```

- Question: which line(s) are invalid? Why?

Another disallowed example

- Consider this pointer modification:

```
char *c_ptr = NULL;
c_ptr = 500; /* Point to addr 500 */
*c_ptr = 56; /* Set 500 to 56 */
```

- Why would you want to do this?
  - Everyone in CS 250, raise your hand
- Sometimes you really need to commit these brutal acts of type insensitivity.
- So far, we've shielded you from the knowledge of this kind of violence in this class
  - No longer!

Casts

- You can use a language construct called a cast to tell the compiler “Hey, trust me on this assignment. I know what I’m doing.”
- Example:
```
char *c_ptr = NULL;
c_ptr = (char *) 500; /* Point to 500 */
*c_ptr = 56;          /* Set 500 to 56 */
```
- The highlighted part is the cast. This is called typecasting or casting a value to a different type
- Many times there is no data conversion taking place!
  - A cast just tells the compiler to allow the assignment
  - Conversions still happen with integral and float types

Allowing the first example…

- Consider the first pointer assignment example with casts inserted in the necessary locations:

```
char *c_ptr = NULL;
int *i_ptr = NULL;
int *i_arr = malloc(sizeof(int) * 4);
c_ptr = (char *) i_arr;
i_ptr = (int *) c_ptr;
i_ptr[1] = 7;
```

- Question: Will this code properly set the value of i_ptr[1] to 7?

Cast syntax

- You can generally cast a value to any type
- A cast always consists of a type enclosed within parentheses – all within an expression. E.g.:
```
x = (int) y;
x = (int) a + (int) b;
x = (int) (a + b);
s = (const struct something * const *) y;
x = ((const struct something *) y)->value;
```
- Sometimes it looks very messy…

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
  - This isn’t guaranteed to be portable
- 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 serve as a conveyor of opaque data or data whose type is not yet known
- Example: our friend, 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 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);
  }
}
```

For next lecture

- Topics for next time:
  - Callbacks
  - Efficiency Issues

Boiler Up!