CS 240: Programming in C

Lecture 20: Preprocessor, Casts, Callbacks

PURDUE UNIVERSITY

Announcements

- Homework 10 released
- Homework 9 due this Wednesday

Announcements

- C23 standard was published on 10/31
    - New functions memccpy(), strdup(), strndup()
    - New preprocessor directives (e.g. #elifdef, #elifndef, #embed)
    - New keywords (bool, true, false, nullptr, constexpr)
    - Many other changes
- We’ll still use C17 in this class

The preprocessor

- When a .c file is complied, 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 a special note of it
  - Or, perhaps, takes other actions
- We’ve seen only two preprocessors directives so far:
  - #define and #include

#include

- #include 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

PURDUE UNIVERSITY

Example of #include

/home/may5/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/may5/x.h

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

Example of #include

/home/may5/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/may5/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

- An 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 definition, the debug statements are no longer compiled

More preprocessor directives

- An 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 definition, the debug statements are no longer compiled

More preprocessor directives

- More flexible directives

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

- You can also have mathematical expressions

```
#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
```

- 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);
```

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
  - Fewer scope issues

```
#define CLOSE_FILE(fp) \
  fclose(fp); \
  fp = NULL;
```

  - If CLOSE_FILE were a function, fp would need to be a double pointer

Why macros?

- Passed arguments can be of any type

```
#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, double, even chars. They all work.
  - A function called max() would not be this flexible

Macro pitfalls

- What’s wrong here?

```
#define CLOSE_FILE(fp) \
  fclose(fp); \
  fp = NULL;
```

```
int ret = fscanf(fp, “%d”, &n);
if (ret < 1)
  CLOSE_FILE(fp);
else
  printf(“%d\n”, n);
```

Macro pitfalls

- The previous slide expands to:

```
int ret = fscanf(fp, “%d”, &n);
if (ret < 1)
  fclose(fp);
  fp = NULL;
else
  printf(“%d\n”, n);
```

No braces, won’t compile!

Macro pitfalls

- Let’s change the macro… does it fix the problem?

```
#define CLOSE_FILE(fp) \
  { \
    fclose(fp); \
    fp = NULL; \
  }
```

```
int ret = fscanf(fp, “%d”, &n);
if (ret < 1)
  CLOSE_FILE(fp);
else
  printf(“%d\n”, n);
```

PURDUE UNIVERSITY

Macro pitfalls

- The previous slide expands to:

```
int ret = fscanf(fp, “%d”, &n);
if (ret < 1)
  {
    fclose(fp);
    fp = NULL;
  };
else
  printf(“%d\n”, n);
```

Extra semicolon, won’t compile!

Macro pitfalls

- Try not to use a semicolon after the macro, or...
- We can use this weird trick to “ignore” the semicolon

```
#define CLOSE_FILE(fp) \
  do { \
    fclose(fp); \
    fp = NULL; \
  } while(0)
```

Note: no semicolon

- Loop runs exactly once

Macro pitfalls

- The previous slide expands to:

```
int ret = fscanf(fp, “%d”, &n);
if (ret < 1)
  do {
    fclose(fp);
    fp = NULL;
  } while(0);
else
  printf(“%d\n”, n);
```

Macro pitfalls

- Beware of duplicating side-effects

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

```
int foo(int z) {
  printf(“%d\n”, z);
  return z + 1;
}
```

```
int next = MAX(x + y, foo(z));
```

- Which expands to:

```
int next = ((x + y) > (foo(z)) ? (x + y) : (foo(z)));
```

Other preprocessor tricks

- We could spend a lot of time looking at the nuances of the preprocessor
- Consider the following:

```
printf("The date is %s\n", __DATE__);
printf("The time is %s\n", __TIME__);
printf("This line is number %d in file %s\n",
       __LINE__, __FILE__);
```

Date and time of compilation (not runtime)

- Most of the preprocessor features are for advanced software development practices

PURDUE UNIVERSITY

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

```
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;
```

- Which of these lines causes a compiler error?

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 ever want to do this?
- Sometimes it’s necessary
    - e.g., embedded systems, drivers
    - Make sure you know what you’re doing

Casts

- You can use a **cast** to tell the compiler, “Trust me on this assignment. I know what I’m doing.”

```
char *c_ptr = NULL;
c_ptr = (char *)500; /* Point to addr 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 between integer 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;
```

- 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

```
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)
- 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);
}
```

PURDUE UNIVERSITY

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);
  }
}
```

For next lecture

- Study and practice the examples in the slides!
- Work on Homeworks 9 and 10

Slides

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