CSE 374
Programming Concepts & Tools

Hal Perkins

Spring 2022

Lecture 13 – C: Multi-File Programs, Header Files & The Rest of the Preprocessor

Administrivia

- HW5 out now, due in a week
  - Short demo today

- HW6 – multiple parts with a partner
  - Pick a partner by early next week. Partner info *must* be submitted online by 11 pm next Wednesday (details on how/where later this week)

The story so far…

- We've looked at the basics of the preprocessor
  - #include to access declarations in header files
  - #define for symbolic constants
- Now:
  - More details; where it fits
  - Multiple source and header files
  - A bit about macros (somewhat useful, somewhat a warning)

Multi-File C Programs

- Our first C programs had a single file with multiple functions, one named `main` where execution starts
- Real programs need to be split into multiple source files (modules) that can be built/tested independently and linked together to build the final program.
- Modularity: the degree to which components of a system can be separated and recombined
  - "Loose coupling" and "separation of concerns"
  - Modules can be developed independently
  - Modules can be re-used in different projects

C Header Files and Modularity

- Header: a C file whose only purpose is to be #include'd
- Generally has a filename .h extension
- Holds the variables, types, and function prototype declarations that make up the interface to a module
- Main Idea: split program into modules of .h/.c pairs of files
  - File name.c implements a module that has an associated name.h that specifies it
  - name.h declares the interface to that module
  - Other modules can use name by #include-ing name.h
  - They should assume as little as possible about the implementation in name.c

Example: program using a linked list

```c
#include <stdlib.h>
#include <assert.h>
#include "ll.h"

Node* Push(Node* head,
           int element) {
  ... // implementation here
}
```

ll.c

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head,
           int element);
```

ll.h

```c
#include "ll.h"

int main(int argc, char** argv) {
  Node* list = NULL;

  list = Push(list, 17);
  list = Push(list, 42);

  ...

  return EXIT_SUCCESS;
}
```

example_ll_customer.c

Compiling the Program

- Four (well, really three) steps
  1. Compile example_ll_customer.c to get example_ll_customer.o
  2. Compile ll.c to get ll.o
  3. Link .o files to get example_ll_customer executable
  4. Test, debug, …

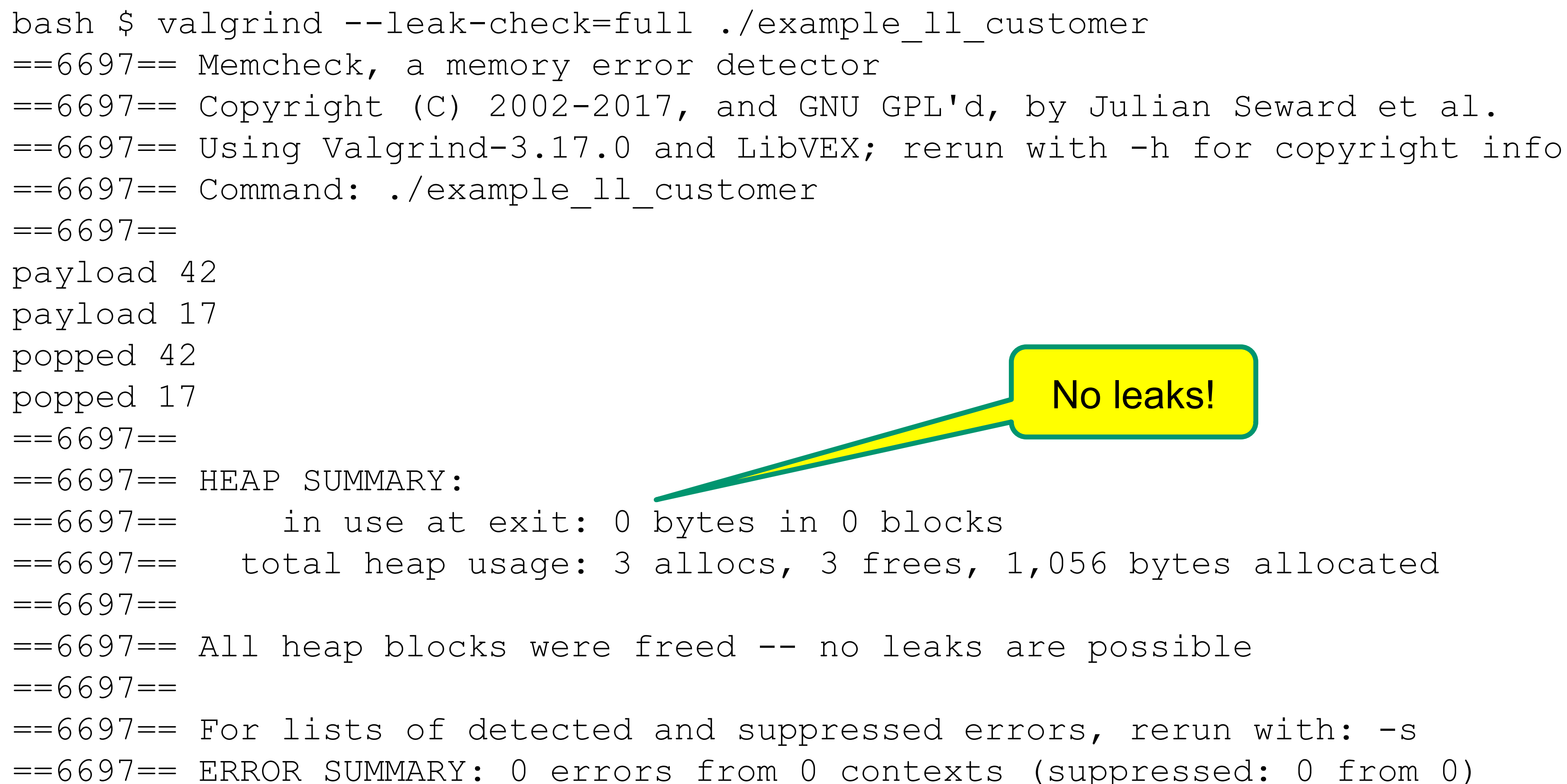
```
bash $ gcc -Wall -g -std=c17 -c -o example_ll_customer.o example_ll_customer.c
bash $ gcc -Wall -g -std=c17 -c -o ll.o ll.c
bash $ gcc -Wall -g -std=c17 -o example_ll_customer ll.o example_ll_customer.o
bash $ ./example_ll_customer
payload 42
payload 17
popped 42
popped 17
```

-c compile .c to .o and stop without linking

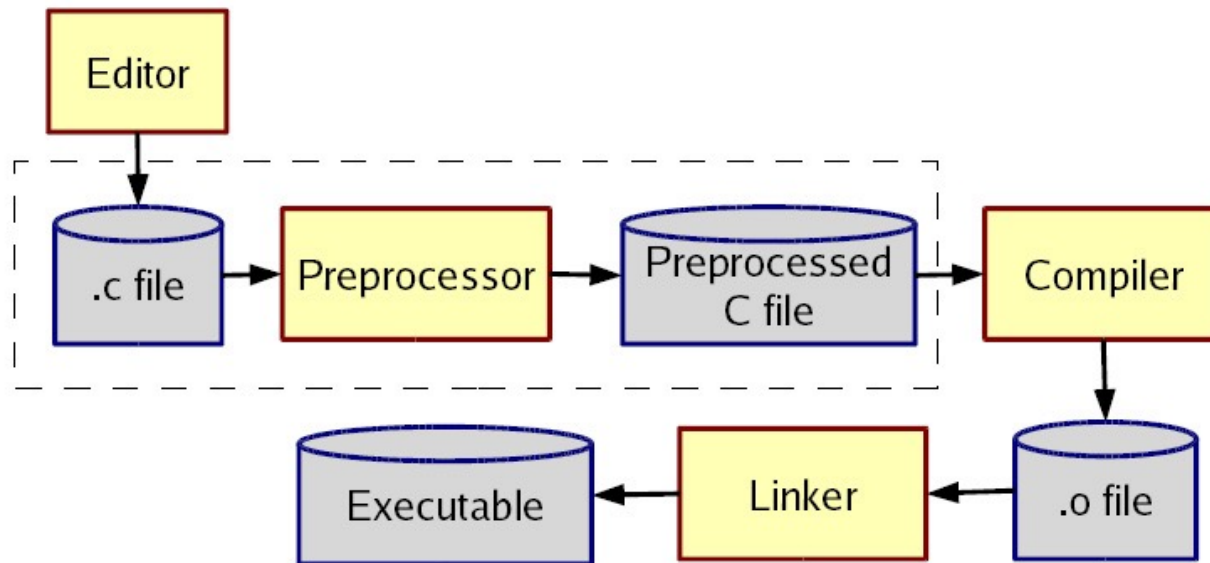
link .o files to get executable program

While we're at it…

- Check for memory leaks : valgrind
  - Super useful tool for hw5



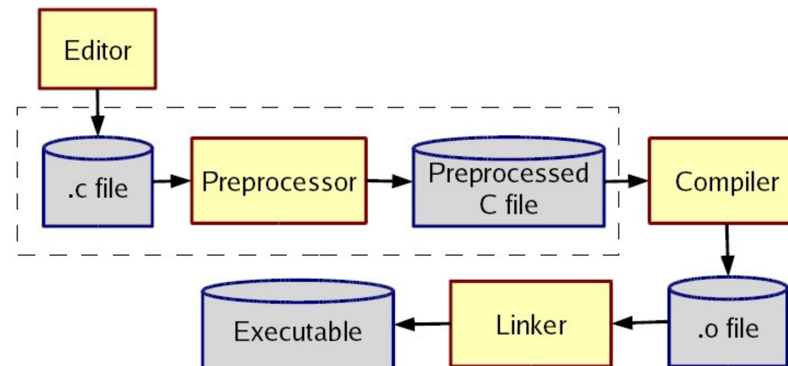
The compilation picture



gcc does all this for you (reminder)

- -E to only preprocess; result on stdout (rare)
- -c to stop with .o (common for individual files in larger program)

More about multiple files



Typical usage:

- Preprocessor #include to read file containing declarations describing code
- Linker combines your .o files *and* other code
  - By default, the "standard C library"
  - Other .o and .a files
  - Whole lecture on linking and libraries later…

The preprocessor

- Rewrites your .c file before the compiler gets at the code.
  - Lines starting with # tell it what to do
- Can do crazy things (please don't); uncrazy things are:
  1. Including contents of header files
  2. Defining constants and parameterized macros
     - Token-based, but basically textual replacement
     - Easy to misdefine and misuse
  3. Conditional compilation
     - Include/exclude part of a file
     - Example uses: code for debugging, code for particular computers (handling portability issues), "the trick" for including header files only once

File inclusion (review)

#include <hdr.h>

- Search for file hdr.h in “standard include directories” and include its contents in this place
  - Typically lots of nested includes, result not fit for human consumption
  - Idea is simple: declaration of standard library routines are in headers; allows correct use after declaration

#include "hdr.h"

  - Same, but first look in current directory
  - How to break your program into smaller files that can call routines in other files

- gcc -I option: look first in specified directories for headers (keep paths out of your code files) (not needed for 374)

C Module Conventions

Most C projects adhere to the following rules:

- `.h` files only contain declarations, never definitions
- `.c` files never contain prototype declarations for functions that are intended to be exported through the module interface
  - Those function prototype declarations belong in the `.h` file
- NEVER `#include` a `.c` file – only `#include .h` files
- `#include` all of headers you reference, even if another header (accidentally or not) includes some of them
- Any `.c` file with an associated `.h` file (a module) should be able to be compiled into a `.o` file
  - The `.c` file should `#include` the `.h` file; the compiler will check declarations and definitions for consistency

Header file conventions

Conventions: always follow these when writing a header file

1. Give included files names ending in .h; only include these header files. ***Never*** #include a .c source file
2. Do not put functions definitions in a header file; only struct definitions, prototypes (declarations with comments), and other #includes
3. Put all your #includes at the beginning of a file
4. For header file foo.h start it with:

   ```
   #ifndef FOO_H
   #define FOO_H
   ```

   and end it with:

   ```
   #endif
   ```

   (We will learn why very soon)

Simple macros (review)

Symbolic constants and other text

```
#define NOT_PI  22/7
#define VERSION 3.14
#define FEET_PER_MILE  5280
#define MAX_LINE_SIZE 5000
```

- Replaces all matching *tokens* in rest of file
  - Knows where “words” start and end (unlike sed)
  - Has no notion of scope (unlike C compiler)
  - (Rare: can shadow with another #define or use #undef to remove)

Macros with parameters

```
#define TWICE_AWFUL(x) x*2
#define TWICE_BAD(x) ((x)+(x))
#define TWICE_OK(x) ((x)*2)
double twice(double x) { return x+x; }   // best (editorial opinion)
```

- Replace all matching “calls” with “body” but with text of arguments where the parameters are (*just* string substitution)
- Gotchas (understand why!):

  ```
  y=3;  z=4;  w=TWICE_AWFUL(y+z);
  y=7;  z=TWICE_BAD(++y);  z=TWICE_BAD(y++);
  ```

- Common misperception: Macros avoid performance overhead of a function call (maybe true in 1975, not now)
- Macros can be more flexible though (TWICE_OK works on ints and doubles without conversions (which could round))

Justifiable uses

Parameterized macros are generally to be avoided (use functions), but there are things functions cannot do:

```
#define NEW_T(t, howmany)  ((t*)malloc((howmany)*sizeof(t))

#define PRINT(x)  printf("%s:%d %s\n", __FILE__, __LINE__,x)
```

Conditional compilation

```
#ifdef FOO       (matching #endif later in file)
#ifndef FOO      (matching #endif later in file)
#if FOO > 2      (matching #endif later in file)
```

(You can also have a #else inbetween somewhere.)

Simple use:
```
#ifdef DEBUG // do following only when debugging
    printf(...);
#endif
```

Fancier:
```
#ifdef DEBUG // use DBG_PRINT for debug-printing
#define DBG_PRINT(x) printf("%s",x)
#else
#define DBG_PRINT(x) // replace with nothing
#endif
```

- Note: gcc -D FOO makes FOO “defined”

Back to header files

- Now we know what this means:

```
#ifndef SOME_HEADER_H
#define SOME_HEADER_H
... rest of some_header.h ...
#endif
```

- Assuming nobody else defines SOME_HEADER_H (convention), the first #include "some_header.h" will do the define and include the rest of the file, but the second and later will skip everything
  - More efficient than copying the prototypes over and over again
  - In presence of circular includes, necessary to avoid creating duplicate definitions by multiple #includes of the same header file
- So we always do this

C preprocessor summary

- A few easy to abuse features and a bunch of conventions (for overcoming C's limitations).
  - #include (the way you say what other definitions you need; cycles are fine with "the trick")
  - #define (avoids magic constants; parameterized macros have a few justifiable uses; token-based text replacement)
  - #if... (for showing the compiler less code)