Modules and The C Preprocessor

CSE 333 Winter 2020

Instructor: Justin Hsia

Teaching Assistants:

Andrew Hu
Austin Chan
Brennan Stein
Cheng Ni
Cosmo Wang
Diya Joy
Guramrit Singh
Mengqi Chen
Pat Kosakanchit
Rehaan Bhimar
Renshu Gu
Travis McGaha
Zachary Keyes

Administrivia

- Exercise 4 out today and due Friday morning
- Exercise 5 will rely on material covered in Section 2
  - Released Thursday afternoon instead
  - *Much* longer and harder than previous exercises!
- Exercise 6 released on Friday (instead of Monday)
- *Both exercise 5 and 6 are due next Wednesday (1/22)*

- Homework 1 due in a week
  - Advice: be *sure* to read headers carefully while implementing
  - Advice: use git add/commit/push often to save your work

Linked List Code for Memory Diagram

```
Node* Push(Node* head, void* e) {
  Node* n = malloc(sizeof(Node));
  assert(n != NULL);
  n->element = e;
  n->next = head;
  return n;
}
```

manual_list_void.c

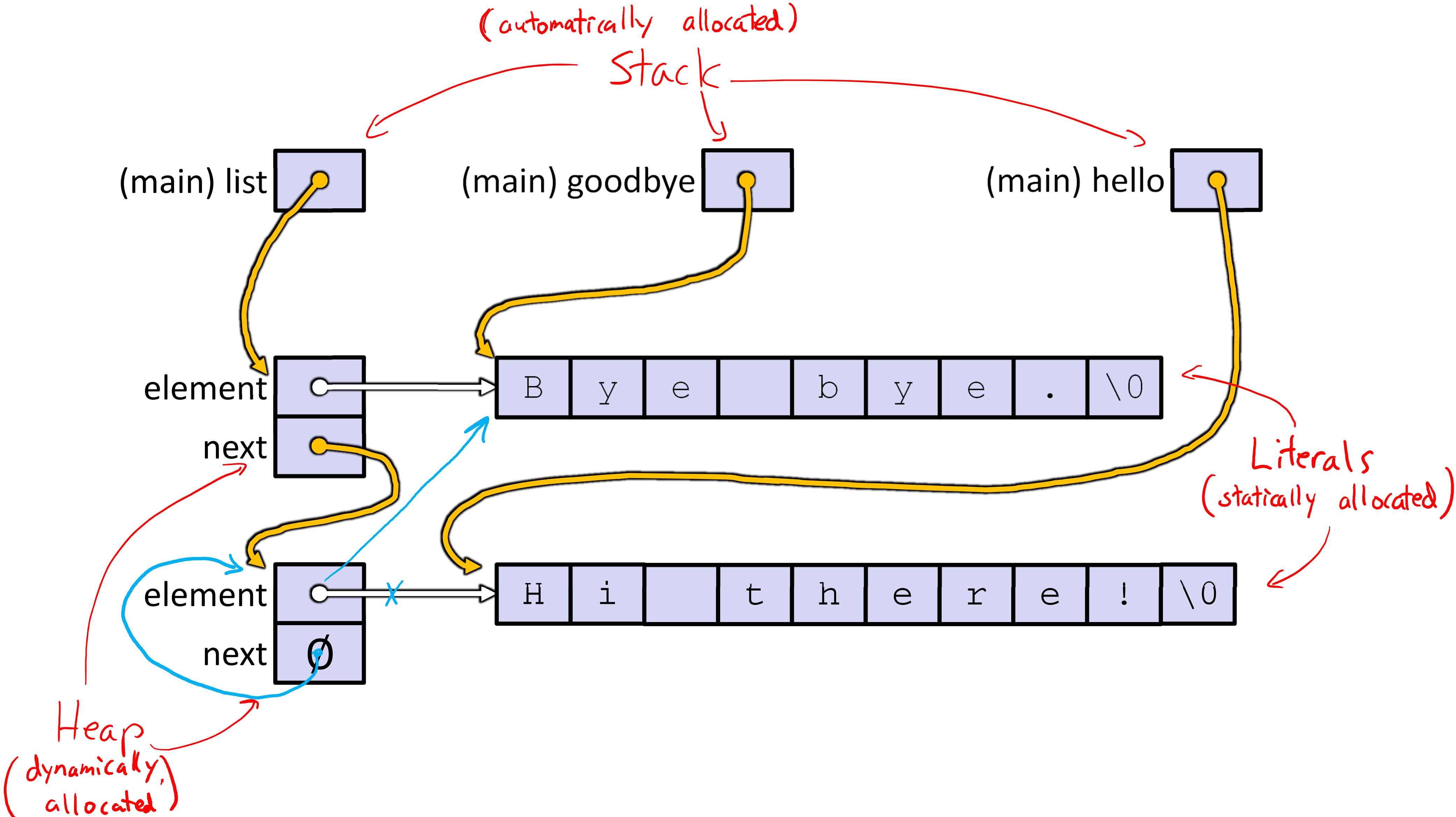
```
typedef struct node_st {
  void* element;
  struct node_st* next;
} Node;

Node* Push(Node* head, void* e);

int main(int argc, char** argv) {
  char* hello = "Hi there!";
  char* goodbye = "Bye bye.";
  Node* list = NULL;

  list = Push(list, (void*) hello);
  list = Push(list, (void*) goodbye);
  return EXIT_SUCCESS;
}
```

Resulting Memory Diagram



What would happen if we execute `*(list->next) = *list`?

structs are copied by value

Something's Fishy…

- A (benign) memory leak!

```
int main(int argc, char** argv) {
  char* hello = "Hi there!";
  char* goodbye = "Bye bye.";
  Node* list = NULL;

  list = Push(list, (void*) hello);
  list = Push(list, (void*) goodbye);
  return EXIT_SUCCESS;
}
```

- Try running with Valgrind:

```
bash$ gcc -Wall -g -o manual_list_void manual_list_void.c
bash$ valgrind --leak-check=full ./manual_list_void
```

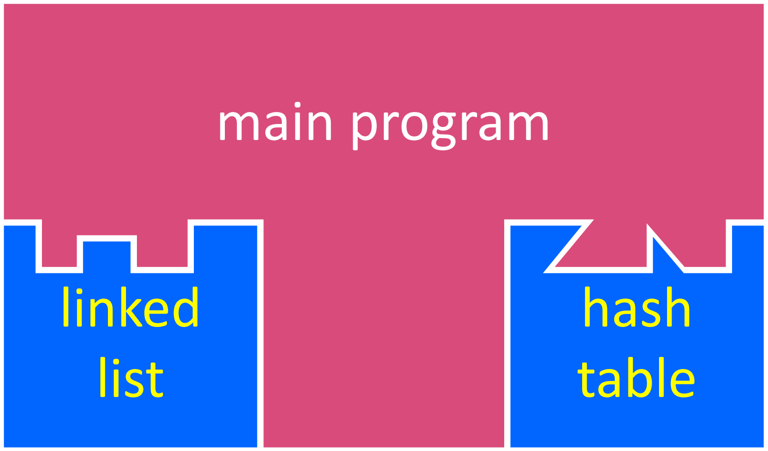
Lecture Outline

- **Structuring Interfaces**
- C Preprocessor and Header Guards
- Visibility of Symbols
  - `extern`, `static`

Multi-File C Programs

- Let's create a linked list *module*
  - A module is a self-contained piece of an overall program
    - Has externally visible functions that customers can invoke
    - Has externally visible `typedefs`, and perhaps global variables, that customers can use
    - May have internal functions, `typedefs`, or global variables that customers should *not* look at
  - Can be developed independently and re-used in different projects

- The module's *interface* is its set of public functions, `typedefs`, and global variables



C Header Files

- Header: a 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 (not definitions)
  - There are <system-defined> and "programmer-defined" headers
    #include <stdio.h> #include "my_header.h"

- Main Idea:
  - Every `name.c` is intended to be a module that has a `name.h`
  - `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`

C Module Conventions (1 of 2)

STYLE TIP

- File contents:
  - `.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
  - Public-facing functions are `ModuleName_functionname()` (GenericLinkedList_Push()) and take a pointer to "`this`" as their first argument
- Including:
  - ***NEVER*** `#include` a `.c` file – only `#include .h` files
  - `#include` all of headers you reference, even if another header (transitively) includes some of them
- Compiling:
  - Any `.c` file with an associated `.h` file should be able to be compiled (together via `#include`) into a `.o` file

C Module Conventions (2 of 2)

STYLE TIP

- Commenting:
  - If a function is declared in a header file (`.h`) and defined in a C file (`.c`), *the header needs full documentation because it is the public specification*
    - Don't copy-paste the comment into the C file (don't want two copies that can get out of sync)
  - If prototype and implementation are in the same C file:
    - School of thought #1: Full comment on the prototype at the top of the file, no comment (or "declared above") on code
    - School of thought #2: Prototype is for the compiler and doesn't need comment; comment the code to keep them together

e.g. 333 project code

Lecture Outline

- Structuring Interfaces
- **C Preprocessor and Header Guards**
- Visibility of Symbols
  - `extern`, `static`

`#include` and the C Preprocessor

- The C preprocessor (`cpp`) is a *sequential* and *stateful* search-and-replace text-processor that transforms your source code before the compiler runs
  - The input is a C file (text) and the output is still a C file (text)
  - It processes the directives it finds in your code (*#directive*)
    - *e.g.* `#include "ll.h"` is replaced by the post-processed content of `ll.h`
      - " " – look in local directory
      - < > – look in library directory
    - *e.g.* `#define PI 3.1415` defines a symbol and replaces later occurrences
      - macro text substitution
    - Several others that we'll see soon…
  - Run automatically on your behalf by `gcc` during compilation

C Preprocessor Example

preprocessor state

FOO	1
BAR	2 + 1

- What do you think the preprocessor output will be?

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

⇒

```
typedef long long int verylong;

int main(int argc, char** argv) {
  int x = 1;
  int y = 2 + 1;
  verylong z = 1 + 2 + 1;
  return 0;
}
```

C Preprocessor Example

- We can manually run the preprocessor:
  - `cpp` is the preprocessor (can also use `gcc -E`)
  - "`-P`" option suppresses some extra debugging annotations

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
bash$ cpp -P cpp_example.c out.c
bash$ cat out.c

typedef long long int verylong;
int main(int argc, char **argv) {
  int x = 1;
  int y = 2 + 1;
  verylong z = 1 + 2 + 1;
  return 0;
}
```

Program Using a Linked List

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

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

ll.c

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

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

ll.h

```c
#include "ll.h"

int main(int argc, char** argv) {
  Node* list = NULL;
  char* hi = "hello";
  char* bye = "goodbye";

  list = Push(list, (void*)hi);
  list = Push(list, (void*)bye);

  ...

  return 0;
}
```

example_ll_customer.c

Node def needed here

Push declaration needed here

Compiling the Program

- Four parts:
  - 1/2) Compile `example_ll_customer.c` into an object file
  - 2/1) Compile `ll.c` into an object file
  - 3) Link both object files into an executable
  - 4) Test, Debug, Rinse, Repeat

```
bash$ gcc -Wall -g -c -o example_ll_customer.o example_ll_customer.c
bash$ gcc -Wall -g -c -o ll.o ll.c
bash$ gcc -g -o example_ll_customer ll.o example_ll_customer.o
bash$ ./example_ll_customer
Payload: 'yo!'
Payload: 'goodbye'
Payload: 'hello'
bash$ valgrind -leak-check=full ./example_ll_customer
... etc ...
```

But There's a Problem with #include

- What happens when we compile `foo.c`?

```c
struct pair {
  int a, b;
};
```

pair.h

```c
#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);
```

util.h

```c
#include "pair.h"
#include "util.h"

int main(int argc, char** argv) {
  // do stuff here
  ...
  return 0;
}
```

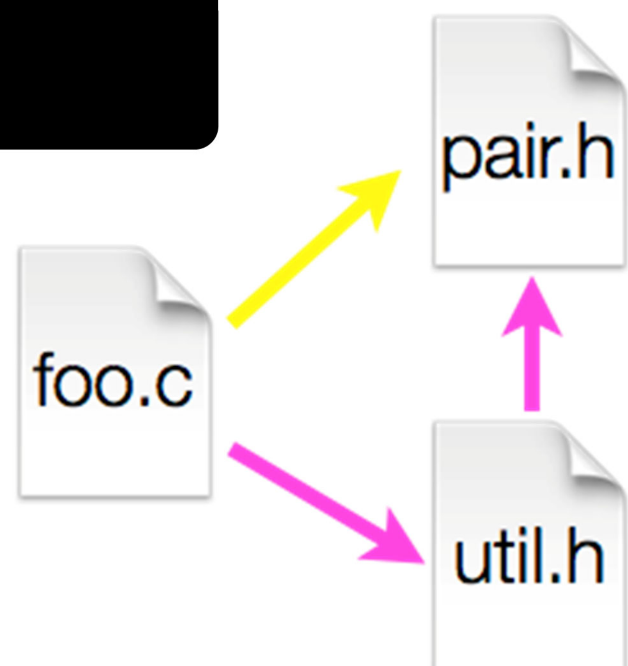
foo.c

A Problem with `#include`

- What happens when we compile `foo.c`?

```
bash$ gcc -Wall -g -o foo foo.c
In file included from util.h:1:0,
                 from foo.c:2:
pair.h:1:8: error: redefinition of 'struct pair'
 struct pair { int a, b; };
        ^
In file included from foo.c:1:0:
pair.h:1:8: note: originally defined here
 struct pair { int a, b; };
        ^
```

- `foo.c` includes `pair.h` twice!
  - Second time is indirectly via `util.h`
  - Struct definition shows up twice
    - Can see using `cpp`



STYLE TIP

Preprocessor Tricks: Header Guards

- A standard C Preprocessor trick to deal with this
  - Uses macro definition (`#define`) in combination with conditional compilation (`#ifndef` and `#endif`)

```c
#ifndef PAIR_H_
#define PAIR_H_

struct pair {
  int a, b;
};

#endif  // PAIR_H_
```

pair.h

```c
#ifndef UTIL_H_
#define UTIL_H_

#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);

#endif  // UTIL_H_
```

util.h

foo.c

```c
#include "pair.h"
#include "util.h"

int main(int argc, char** argv) {
```

Preprocessor Tricks: Constants

STYLE TIP

- A way to deal with "magic constants"

```
int globalbuffer[1000];

void circalc(float rad,
             float* circumf,
             float* area) {
  *circumf = rad * 2.0 * 3.1415;
  *area = rad * 3.1415 * 3.1415;
}
```

Bad code
(littered with magic constants)

```
#define BUFSIZE 1000
#define PI 3.14159265359

int globalbuffer[BUFSIZE];

void circalc(float rad,
             float* circumf,
             float* area) {
  *circumf = rad * 2.0 * PI;
  *area = rad * PI * PI;
}
```

Better code

Preprocessor Tricks: Macros

- You can pass arguments to macros

```c
#define ODD(x) ((x) % 2 != 0)

void foo() {
  if ( ODD(5) )
    printf("5 is odd!\n");
}
```

cpp →

```c
void foo() {
  if ( ((5) % 2 != 0) )
    printf("5 is odd!\n");
}
```

treated as just text

- Beware of operator precedence issues!
  - Use parentheses

```c
#define ODD(x) ((x) % 2 != 0)
#define WEIRD(x) x % 2 != 0

ODD(5 + 1);

WEIRD(5 + 1);
```

cpp →

```c
((5 + 1) % 2 != 0);

5 + 1 % 2 != 0;
```

Preprocessor Tricks: Defining Tokens

- Besides `#define`s in the code, preprocessor values can be given as part of the `gcc` command:

```
bash$ gcc -Wall -g -DTRACE -o ifdef ifdef.c
```

-D define
-U undefine

- assert can be controlled the same way – defining `NDEBUG` causes assert to expand to "empty"
  - It's a macro – see `assert.h`

```
bash$ gcc -Wall -g -DNDEBUG -o faster useassert.c
```

Preprocessor Tricks: Conditional Compilation

#ifdef = "if defined"
#ifndef = "if not defined"

- You can change what gets compiled
  - In this example, `#define TRACE` before `#ifdef` to include debug `printf`s in compiled code

```
#ifdef TRACE
#define ENTER(f) printf("Entering %s\n", f);
#define EXIT(f)  printf("Exiting  %s\n", f);
#else
#define ENTER(f)
#define EXIT(f)
#endif

// print n
void pr(int n) {
  ENTER("pr");
  printf("\n = %d\n", n);
  EXIT("pr");
}
```

ifdef.c

Polling Question

- What will happen when we try to compile and run?
  - Vote at http://PollEv.com/justinh

```
bash$ gcc -Wall -DFOO -DBAR -o condcomp condcomp.c
bash$ ./condcomp
```

FOO and BAR are defined

A. Output "333"

B. Output "334"

C. Compiler message about EVEN

D. Compiler message about BAZ

E. We're lost…

```
#include <stdio.h>
#ifdef FOO          ← yes
#define EVEN(x) !(x%2)
#endif
#ifndef DBAR        ← no
#define BAZ 333
#endif

int main(int argc, char** argv) {
  int i = EVEN(42) + BAZ;
  printf("%d\n",i);
  return 0;
}
```

!(42%2) evaluates to !0 = 1

333

Extra Exercise #1

- Implement and test a binary search tree
  - https://en.wikipedia.org/wiki/Binary_search_tree
    - Don't worry about making it balanced
  - Implement key insert() and lookup() functions
    - Bonus: implement a key delete() function
  - Implement it as a C module
    - `bst.c`, `bst.h`
  - Implement `test_bst.c`
    - Contains main() and tests out your BST

Extra Exercise #2

- Implement a Complex number module
  - `complex.c`, `complex.h`
  - Includes a typedef to define a complex number
    - a + bi, where a and b are `doubles`
  - Includes functions to:
    - add, subtract, multiply, and divide complex numbers
  - Implement a test driver in `test_complex.c`
    - Contains `main()`

Resulting Memory Diagram