CS 50:
Software Design and Implementation

Modular data structures

Today we will create something similar to an ADT from CS 10

```
 8 public class SinglyLinked<T> implements SimpleList<T> {
 9     private Element head;    // front of the linked list
10     private int size;        // # elements in the list
11
12     /**
13      * The linked elements in the list: each has a piece of data and
14      */
15     private class Element {
16         private T data;
17         private Element next;
18
19         private Element(T data, Element next) {
20             this.data = data;
21             this.next = next;
22         }
23     }
24
25     public SinglyLinked() {
26         head = null;
27         size = 0;
28     }
29
30     public int size() {
31         return size;
32     }
33
34     /**
35      * Helper function, advancing to the nth Element in the list and
36      * (exception if not that many elements)
37      */
38     private Element advance(int n) throws Exception {
39         Element e = head;
40         while (n > 0) {
41             // Just follow the next pointers
42             e = e.next;
43             if (e == null) throw new Exception("invalid index");
44             n--;
45         }
46         return e;
47     }
```

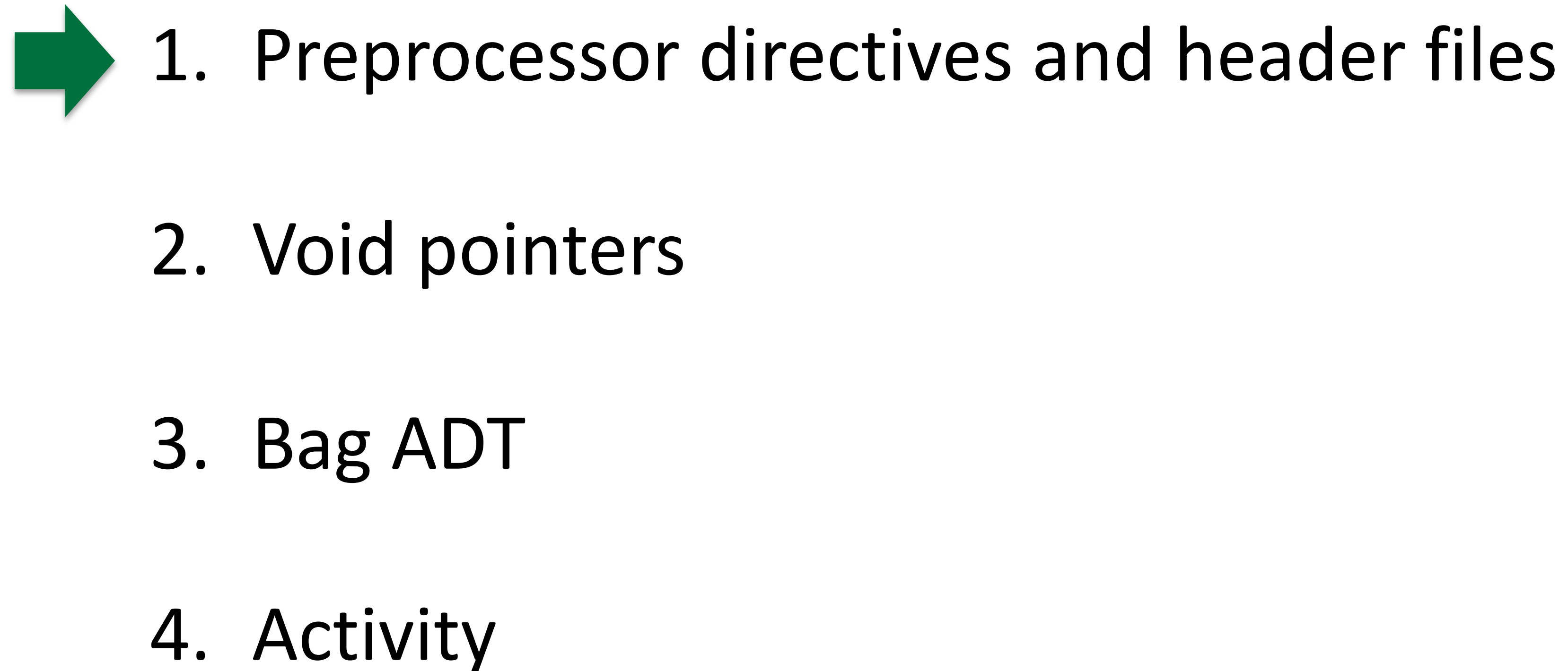
In CS 10 we created a Java class to implement a linked list

We used Java's generics so the list could hold any type of object

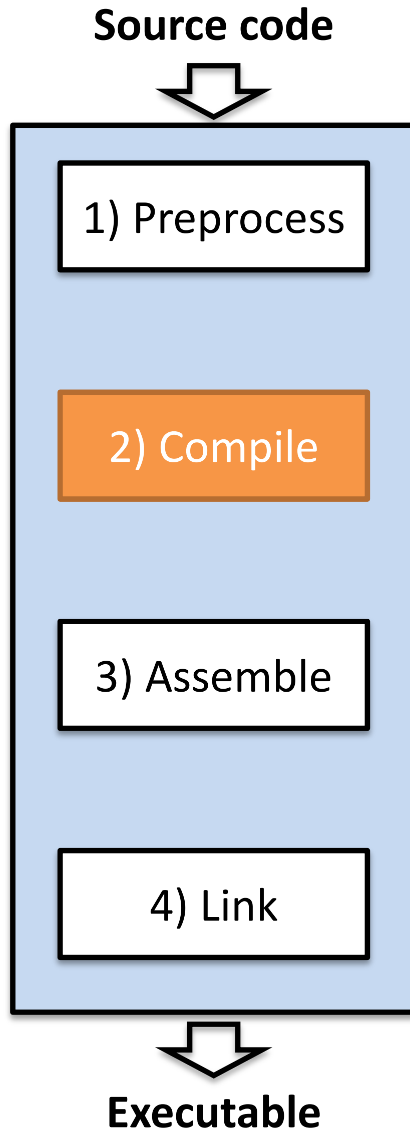
C doesn't have generics

We can approximate the behavior by using void pointers

Agenda



The compiler reads down the code during compilation process



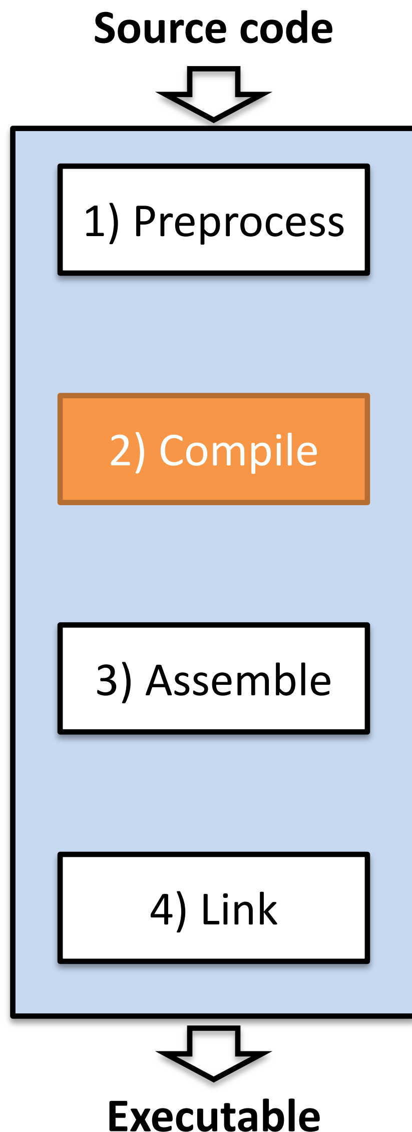
main1.c

```
#include<stdio.h>

int func(int x) {
        return x*x;
}

int main() {
        printf("%d\n",func(5));
        return 0;
}
```

```
$ mygcc -o main main1.c
$ ./main
25
```

Works as expected because compiler reads down code
Finds `func` declaration before `func` call

Calling a function before it is declared is a problem!



func declared after main

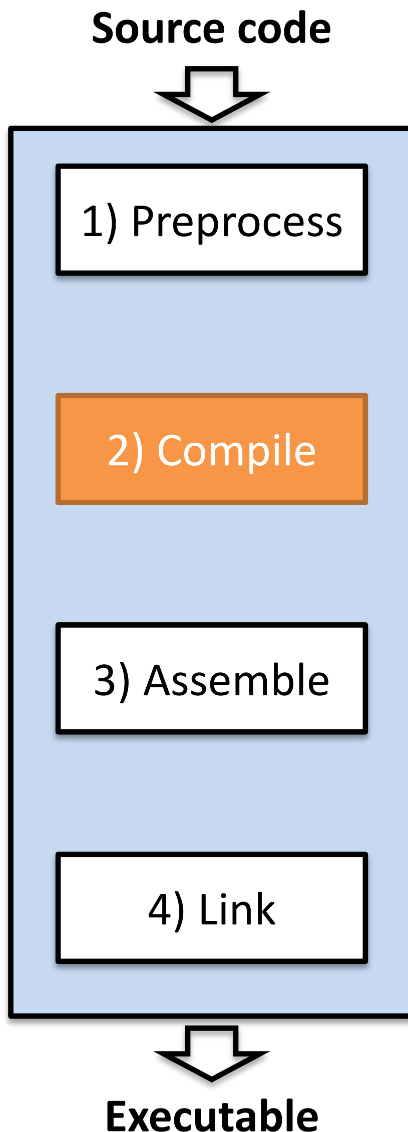
main2.c

```c
#include<stdio.h>

int main() {
        printf("%d\n",func(5));
        return 0;
}

int func(int x) {
        return x*x;
}
```

```
$ mygcc -o main main.c
main.c: In function ‘main’:
main.c:6:16: warning: implicit declaration of function ‘func’; did you mean ‘putc’? [-Wimplicit-function-declaration]
  printf("%d\n",func(5));
                ^~~~
                putc
```

Causes compilation error because compiler does not know about func when it is called

Declare functions before they are called



Declare func so compiler knows about it

main3.c

```
#include<stdio.h>

int func(int);

int main() {
        printf("%d\n",func(5));
        return 0;
}

int func(int x) {
        return x*x;
}
```

Define func

Do not *need* to give parameter names, just parameter type

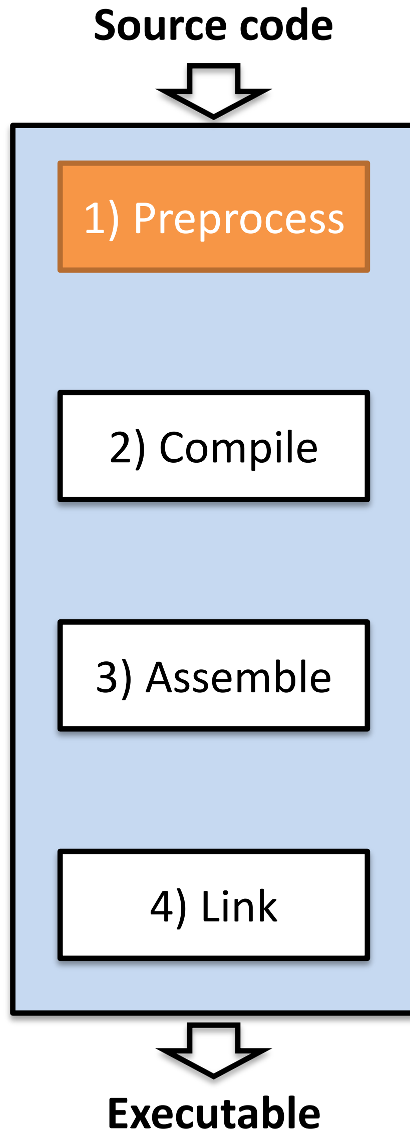
Can provide parameter names if you want to

Sometimes called a *function prototype*

```
$ mygcc -o main main3.c
$ ./main
25
```

Compiles because we told the compiler when we declared func that we would later define it

Header files can declare functions without defining them; expand in preprocess step



main4.h has function declaration

main4.h
```
int func(int x);
```

#include expands contents of file main4.h in this location

Just a text substitution, as if you typed the contents of main4.h here
`int func(int x);`

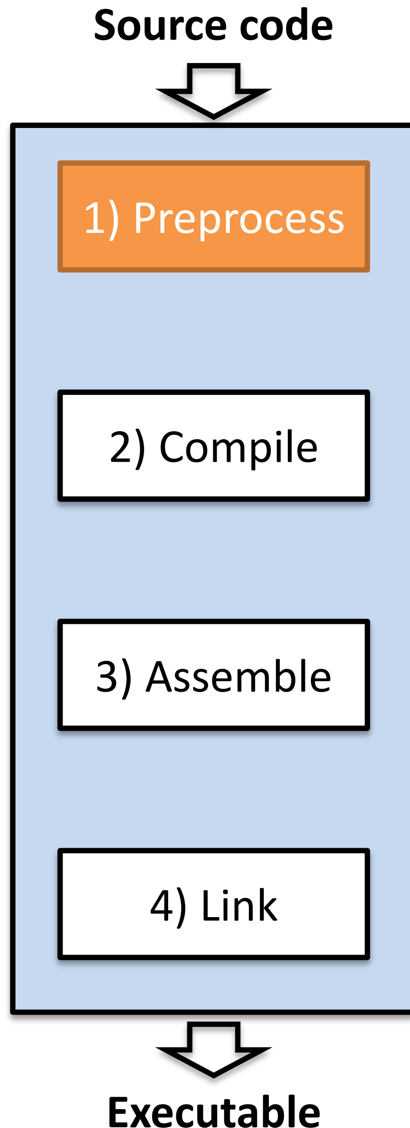
Declares func before func is called so compiles fine

main4.c
```
#include<stdio.h>
#include "main4.h"

int main() {
        printf("%d\n",func(5));
        return 0;
}

int func(int x) {
        return x*x;
}
```

Function definition

```
$ mygcc -o main main4.c
$ ./main
25
```

Header files can declare functions without defining them; expand in preprocess step



main4.h

```
int func(int x);
```

main4.c

```
#include<stdio.h>
#include "main4.h" int func(int x);

int main() {
        printf("%d\n",func(5));
        return 0;
}

int func(int x) {
        return x*x;
}
```

As if we had this after the preprocessor step is run (the line `#include "main4.h"` is struck through and replaced by `int func(int x);`)

Can call function before definition

```
$ mygcc -o main main4.c
$ ./main
25
```

If main4.h declared other functions, they would show up here also

The whole header file is copied here

Other programs can include main4.h
They will also get main4.h's declarations
Header tells the compiler we will define the function later, possibly in another file!

Header file rules of thumb

1. Single .c files do not need a .h file (although you can provide one, some people always do)
2. Break large programs into modules (.c and .h) files that define clear functionality. Compile these together with other modules that provide different functionality (e.g., gcc file1.c file2.c)
3. Everything in a .h file should be used by two or more .c files
4. Put in .h files:
   - Function prototypes (declares but doesn't define function)
   - Custom data types (structs or enums)
   - Anything that defines a type but does not allocate memory (e.g., typedef)
5. Do not put in .h files
   - Anything that allocates memory such as variable declarations
   - Function definitions (e.g., code)

You can think of header file *a little* like an interface in Java

If you have a function outsiders should not call, don't put it in the .h file
Mark function as static in .c file

Adapted from *Learning C* by Jeff Szuhay. Packt, 2020.

We will create programs that can be used by other programs like an ADT in Java

Data structures

Behavior	linked list	bag	set	counters	hashtable
stores an item	yes	yes	yes	no	yes
uses a key	no	no	yes	yes	yes
keeps items in order	yes	no	no	no	no
retrieval	first item	any item	by key	by key	by key
insertion of duplicates	allowed	allowed	error	increment count	error

Notice:
- a *linked list* keeps items in order, but a *bag* or a *set* does not
- a *set* and *hashtable* allow you to retrieve a specific item (indicated by its key) whereas a *bag* might return any item
- because the *bag* and *list* don't distinguish among items they store, they can hold duplicates; the others cannot
- the *counters* data structure maintains a set of counters, each identified by a *key*, but it stores no *items*. Instead, it keeps a counter for each key; inserting a duplicate key increments the counter.

We will create bag.c and bag.h to implement the bag functionality
Other programs can use this code to take advantage of the functionality
Other programs can `#include "bag.h"` to get access to bag's functions
We will use this idea extensively in the Tiny Search Engine project

Prevent repeat definitions when multiple programs include same headers

bag.h

```
#ifndef __BAG_H
#define __BAG_H

/*****global data types****/
typedef struct bag bag_t; // hide the bag structure fro

/***** functions ****/

/* Create a new (empty) bag; return NULL if error. */
bag_t* bag_new(void);

/* Add new item to the bag; a NULL bag is ignored; a NULL item is ignored. */
void bag_insert(bag_t *bag, void *item);

/* Return any data item from the bag; return NULL if bag is NULL or empty. */
void* bag_extract(bag_t *bag);

/* Print the whole bag; provide the output file and func to print each item.
 * If fp==NULL; do nothing. If bag==NULL, print (null).
 * If itemprint==NULL, print nothing for each item.
 */
void bag_print(bag_t *bag, FILE *fp,
               void (*itemprint)(FILE *fp, void *item));

/* Delete the whole bag; ignore NULL bag.
 * Provide a function that will delete each item (may be NULL).
 */
void bag_delete(bag_t *bag, void (*itemdelete)(void *item));

#endif // __BAG_H
```

Checks to see if __BAG_H has been defined

Define if not already defined (notice n in `ifndef`)

Called "header guards"

Now multiple modules can include bag.h and get these function declarations as if they were typed into those module

Compile using `mygcc bag.c <other progs>`

`Other progs can #include "bag.h"`

End of definition

Tip:

Don't start names with single underscore (e.g., '_')

C uses single underscore internally, can cause problems

Either don't use underscore or use two

You can also use #ifdef and #ifndef in code for conditional compilation

define_test.c

```
#include <stdio.h>

int main() {
        #ifdef TESTING
                puts("Got TESTING");
        #else
                puts("Did NOT get TESTING");
        #endif
        printf("Prints regardless\n");

        return 0;
}
```

Preprocess checks to see if `TESTING` is defined
Includes following code in compilation step if defined, includes code following `else` in compilation step if not

TESTING not defined
`puts ("Did NOT get TESTING")` is passed to compilation step

```
$ mygcc define_test.c
$ ./a.out
Did NOT get TESTING
Prints regardless

$ mygcc define_test.c -DTESTING
$ ./a.out
Got TESTING
Prints regardless
```

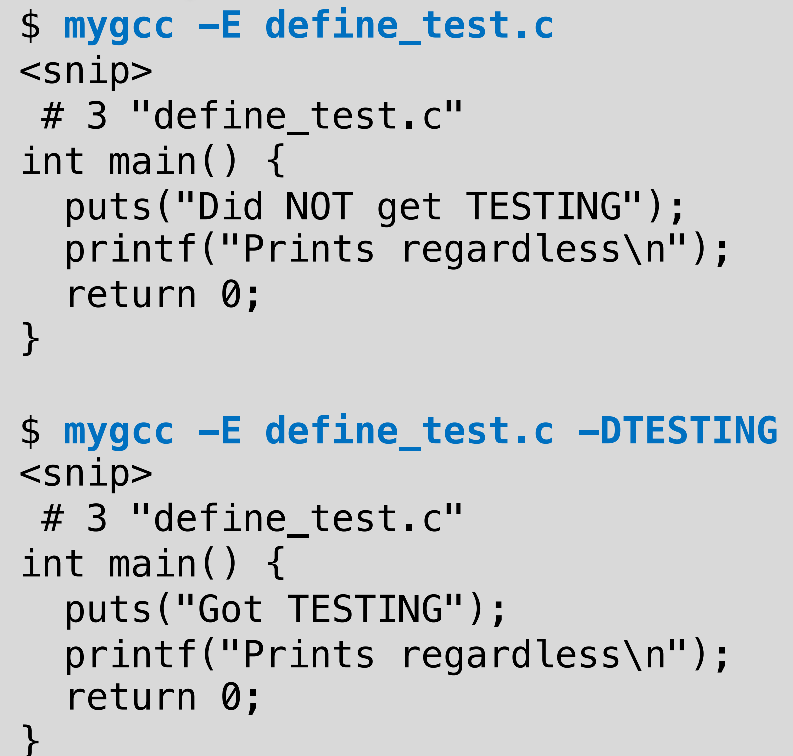
-D flag defines TESTING
`puts ("Got TESTING") is passed` to compilation step

Double check conditional compilation worked using the –E flag

define_test.c

```
#include <stdio.h>

int main() {
        #ifdef TESTING
                puts("Got TESTING");
        #else
                puts("Did NOT get TESTING");
        #endif
        printf("Prints regardless\n");

        return 0;
}
```

-E flag tells gcc to output to the console the code sent to the compilation step, then stop



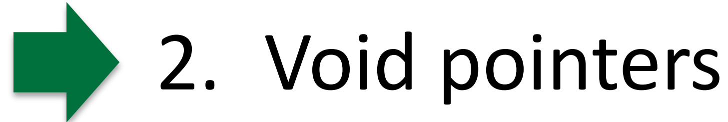
```
$ mygcc –E define_test.c
<snip>
 # 3 "define_test.c"
int main() {
  puts("Did NOT get TESTING");
  printf("Prints regardless\n");
  return 0;
}

$ mygcc –E define_test.c –DTESTING
<snip>
 # 3 "define_test.c"
int main() {
  puts("Got TESTING");
  printf("Prints regardless\n");
  return 0;
}
```

Useful to include debugging print statements only if DEBUG defined

Do not include extraneous print statements in your labs!

Agenda

1. Preprocessor directives and header files



2. Void pointers
3. Bag ADT
4. Activity

Void pointers do not have a type themselves, but can be cast to other types

void_ptr.c

```
15 #include<stdio.h>
16
17 int main() {
18     int x = 42;
19     char *s = "hello";
20     int *xp = &x;
21     void *p = NULL;
22
23     printf("%d %d\n",x,*xp);
24
25     //set void pointer p to to int x's address
26     p = &x;
27     printf("%d\n",*(int *)p); //cast p to int pointer and deference
28
29     //set void pointer p to s's address
30     p = s;
31     printf("%s\n",(char *)p); //cast p to string and deference
32
33
34     return 0;
35 }
```

p is a void pointer
Has no associated type

```
$ mygcc -o void_ptr void_ptr.c
$ ./void_ptr
42 42
42
hello
```

Set p to point to address of x
Then cast p as integer pointer and deference to print whatever p points to as an integer

Now set p to point to string s
Cast p as character pointer and deference to print whatever p points to as a string

Functions pointers act similar to void variable pointers

func_addr.c

```
 8 #include <stdio.h>
 9
10 void print_int(void *val) {
11    int *num = val; //cast void ptr to int ptr
12    if (num != NULL) {
13       printf("%d\n",*num); //deference to print value
14    }
15 }
16
17 void print_string(void *val) {
18    char *s = val; //cast void ptr to char ptr
19    if (s != NULL) {
20       printf("%s\n",s); //print value
21    }
22 }
23
24 int main() {
25    int x = 8;
26    char s[] = "hello";
27
28    //normal call
29    print_int(&x);
30
31    //call using pointer to function
32    void (*func)(void *val);
33    func = print_int;
34    (*func)((void *)&x);
35
36    func = print_string;
37    (*func)((void *)s);
38
39    return 0;
40 }
```

Function that takes a void pointer and prints it as an integer

Function that takes a void pointer and prints it as a string

Normal call passing address of x

```
$ mygcc func_addr.c
$ ./a.out
8
```

Functions pointers act similar to void variable pointers

func_addr.c

```
 8 #include <stdio.h>
 9
10 void print_int(void *val) {
11     int *num = val; //cast void ptr to int ptr
12     if (num != NULL) {
13         printf("%d\n",*num); //deference to print value
14     }
15 }
16
17 void print_string(void *val) {
18     char *s = val; //cast void ptr to char ptr
19     if (s != NULL) {
20         printf("%s\n",s); //print value
21     }
22 }
23
24 int main() {
25     int x = 8;
26     char s[] = "hello";
27
28     //normal call
29     print_int(&x);
30
31     //call using pointer to function
32     void (*func)(void *val);
33     func = print_int;
34     (*func)((void *)&x);
35
36     func = print_string;
37     (*func)((void *)s);
38
39     return 0;
40 }
```

Function that takes a void pointer and prints it as an integer

Function that takes a void pointer and prints it as a string

Normal call passing address of x

Create function pointer, point to `print_int`, and call

Address of a function is just the function name without parenthesis

```
$ mygcc func_addr.c
$ ./a.out
8
8
```

Functions pointers act similar to void variable pointers

func_addr.c

```
 8 #include <stdio.h>
 9
10 void print_int(void *val) {
11     int *num = val; //cast void ptr to int ptr
12     if (num != NULL) {
13         printf("%d\n",*num); //deference to print value
14     }
15 }
16
17 void print_string(void *val) {
18     char *s = val; //cast void ptr to char ptr
19     if (s != NULL) {
20         printf("%s\n",s); //print value
21     }
22 }
23
24 int main() {
25     int x = 8;
26     char s[] = "hello";
27
28     //normal call
29     print_int(&x);
30
31     //call using pointer to function
32     void (*func)(void *val);
33     func = print_int;
34     (*func)((void *)&x);
35
36     func = print_string;
37     (*func)((void *)s);
38
39     return 0;
40 }
```

Function that takes a void pointer and prints it as an integer

Function that takes a void pointer and prints it as a string

```
$ mygcc func_addr.c
$ ./a.out
8
8
hello
```

Normal call passing address of x

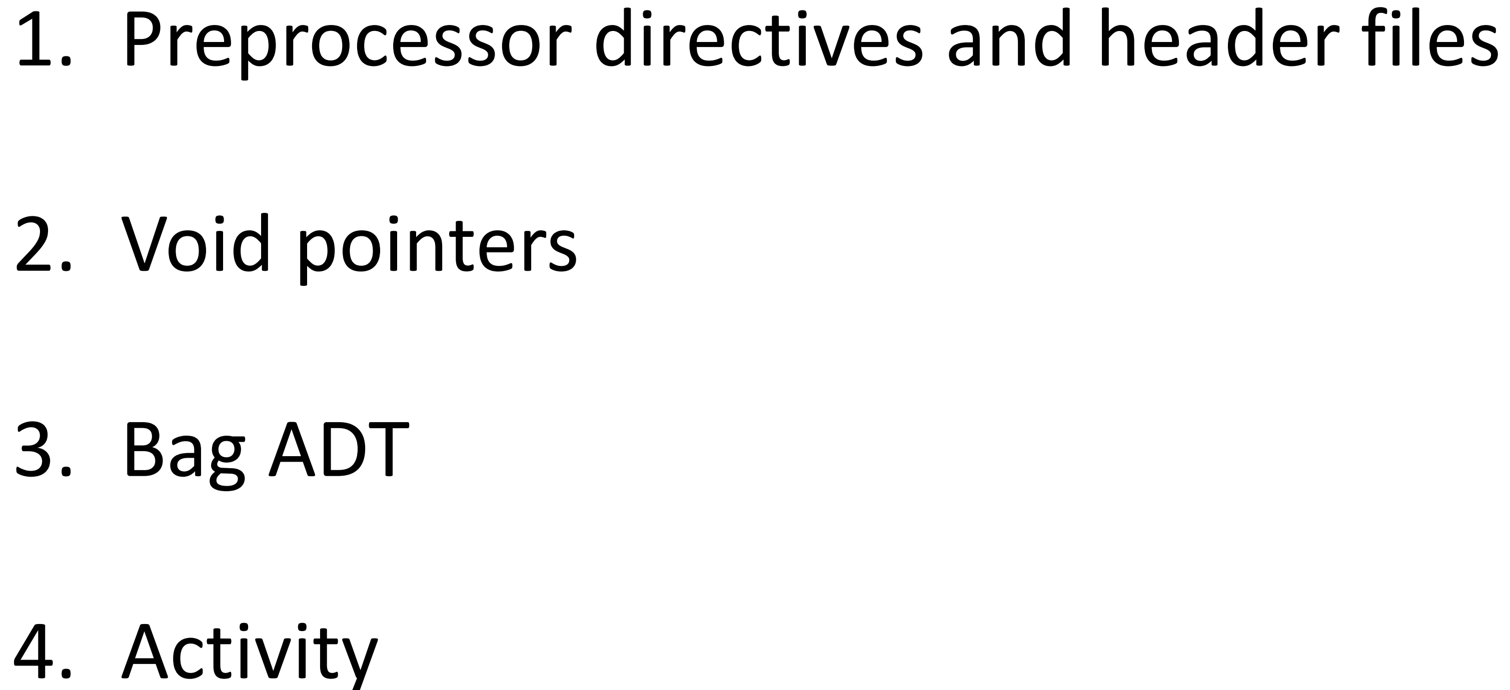
Address of a function is just the function name without parenthesis

Create function pointer, point to `print_int`, and call

Now set func to point to `print_string` and call

We can use this idea to pass functions as parameters!

Agenda



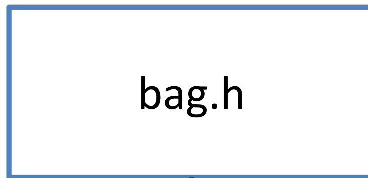
Implement bag ADT functionality, start by declaring functions in .h header file

bag.h

Declares

- bag_new
- bag_insert
- bag_extract
- bag_print
- bag_delete

Define (implement) bag ADT functions in .c file



Declares
- bag_new
- bag_insert
- bag_extract
- bag_print
- bag_delete

bag.c

Includes bag.h
Defines (implements)
- bag_new
- bag_insert
- bag_extract
- bag_print
- bag_delete

Readlinep.h and .c declare and define functions to read strings of arbitrary length

bag.h

Declares
- bag_new
- bag_insert
- bag_extract
- bag_print
- bag_delete

bag.c

Includes bag.h
Defines (implements)
- bag_new
- bag_insert
- bag_extract
- bag_print
- bag_delete

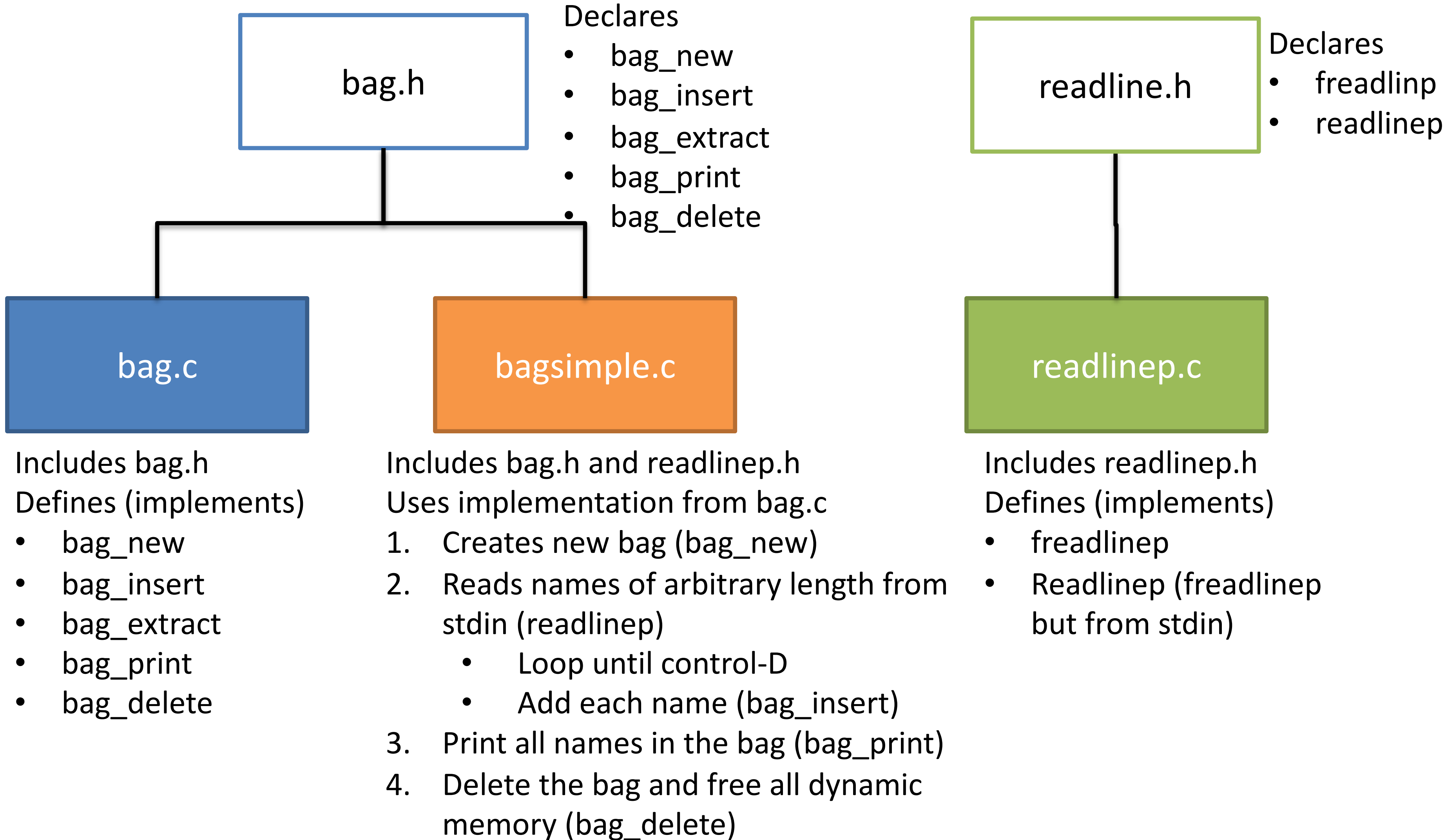
readline.h

Declares
- freadlinp
- readlinep

readlinep.c

Includes readlinep.h
Defines (implements)
- freadlinep
- Readlinep (freadlinep but from stdin)

Use bag ADT and readlinep to create bagsimple application



bagsimple: read names from stdin and stores in a bag, print and free when done

bag.h

Declares
- bag_new
- bag_insert
- bag_extract
- bag_print
- bag_delete

readline.h

Declares
- freadlinp
- readlinep

bag.c

Includes bag.h
Defines (implements)
- bag_new
- bag_insert
- bag_extract
- bag_print
- bag_delete

bagsimple.c

Includes bag.h and readlinep.h
Uses implementation from bag.c
1. Creates new bag (bag_new)
2. Reads names of arbitrary length from stdin (readlinep)
   - Loop until control-D
   - Add each name (bag_insert)
3. Print all names in the bag (bag_print)
4. Delete the bag and free all dynamic memory (bag_delete)

readlinep.c

Includes readlinep.h
Defines (implements)
- freadlinep
- Readlinep (freadlinep but from stdin)

Compile multiple files together to create bagsimple application

```
$ mygcc bag.c readlinep.c bagsimple.c –o bagsimple
$ ./bagsimple
```

bag.h declares functions for use in bag.c and bagsimple.c

bag.h

```
#ifndef __BAG_H
#define __BAG_H


/*****global data types****/
typedef struct bag bag_t; // hide the bag structure from the user

/***** functions ****/

/* Create a new (empty) bag; return NULL if error. */
bag_t* bag_new(void);

/* Add new item to the bag; a NULL bag is ignored; a NULL item is ignored. */
void bag_insert(bag_t *bag, void *item);

/* Return any data item from the bag; return NULL if bag is NULL or empty. */
void* bag_extract(bag_t *bag);

/* Print the whole bag; provide the output file and func to print each item.
 * If fp==NULL; do nothing. If bag==NULL, print (null).
 * If itemprint==NULL, print nothing for each item.
 */
void bag_print(bag_t *bag, FILE *fp,
               void (*itemprint)(FILE *fp, void *item));

/* Delete the whole bag; ignore NULL bag.
 * Provide a function that will delete each item (may be NULL).
 */
void bag_delete(bag_t *bag, void (*itemdelete)(void *item));

#endif // __BAG_H
```

Header guards prevent declaration more than one time

Declares:

- **bag_new**
- **bag_insert**
- **bag_extract**
- **bag_print**
- **bag_delete**

Notice bag doesn't know what kind of items it holds! (uses void pointer)

Bag does not know how to print or free items it holds; we will pass a function via void pointer that does know

bag.h included by

- **bag.c**
- **bagsimple.c**

bag.c implements functions declared in bag.h

bag.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "bag.h"

// local data types
typedef struct bagnode {
        void *item;
        struct bagnode *next;
} bagnode_t;

// global data types
typedef struct bag {
        struct bagnode *head;
} bag_t;

// global functions

/*** bag_new()*****/
bag_t* bag_new(void)
{
        bag_t *bag = malloc(sizeof(bag_t));


        if (bag == NULL) {
                return NULL;
        } else {
                bag->head = NULL;
                return bag;
        }
}
```

Include bag.h to get function declarations Remember including a header file is like typing its contents here

A bagnode holds an item and a pointer to the next item, creating a linked list

typedef means we can just say bagnode_t (t means type) instead of struct bagnode

A bag has a pointer to the head

typedef means we can just say bag_t instead of struct bag

bag_new creates a new bag and returns a pointer to it of type bag_t

Uses malloc so we will have to free it later!

Remember to always check that malloc succeeded!

If malloc success, set head to NULL and return bag struct pointer

bag_insert takes a void pointer to hold any type of item

bag.c

Takes bag parameter, might have multiple bags

Void pointer means any type of item can be stored in the bag

In bagsimple.c it will be strings

```
/*** bag_insert()****/
void bag_insert(bag_t *bag, void *item)
{
        if (bag != NULL && item != NULL) {
                bagnode_t *new = malloc(sizeof(bagnode_t));

                if (new != NULL) {
                        new->item = item;
                        new->next = bag->head;
                        bag->head = new;
                }
        }
}
```

Ensure malloc succeeded! Always check!

Allocate heap space for a new bagnode item

Add new item at front of list

bag_extract removes and returns an item of type void from the bag

bag.c

Takes bag parameter, might have multiple bags

Return NULL if bag or head are NULL

Get bagnode_t to remove (at head)

```
void* bag_extract(bag_t *bag)
{
  if (bag == NULL) {
    return NULL; // bad bag
  } else if (bag->head == NULL) {
    return NULL; // bag is empty
  } else {
    bagnode_t *out = bag->head; // the node to take out
    void *item = out->item;     // the item to return
    bag->head = out->next;      // hop over the node to remove
    free(out);
    return item;
  }
}
```

Get item from badnode_t
Update head to point to next
Free bagnode_t that was at the front
Do not free item yet, it will be returned to caller
The caller will have to free the item

bag_print loops over each item, printing them using pointer to print function

bag.c

Takes bag parameter, might have multiple bags

Print to FILE pointer

```
void bag_print(bag_t *bag, FILE *fp, void (*itemprint)(FILE *fp, void *item) )
{
  if (fp != NULL) {
    if (bag != NULL) {
      fputc('{', fp);
      for (bagnode_t *node = bag->head; node != NULL; node = node->next) {
        // print this node
        if (itemprint != NULL) {  // print the node's item
          (*itemprint)(fp, node->item);
          fputc(',', fp);
        }
      }
      fputs("}\n", fp);
    } else {
      fputs("(null)\n", fp);
    }
  }
}
```

Bag doesn't know what type of items it holds

Caller passes a pointer to a function that does know to print items

Loop over all items in bag

Call print function for each item

Pass in `fp` and `item` to print function

`print_bag` uses that function

bag_delete loops over each item and removes them from the bag

bag.c

Takes bag parameter, might have multiple bags

```
Void bag_delete(bag_t *bag, void (*itemdelete)(void *item) )
{
  if (bag != NULL) {
    for (bagnode_t *node = bag->head; node != NULL; ) {
      if (itemdelete != NULL) {           // if possible...
        (*itemdelete)(node->item);        // delete node's item
      }
      bagnode_t *next = node->next;       // remember what comes next
      free(node);                         // free the node
      node = next;                        // and move on to next
    }


    free(bag);
  }
}
```

Pass in pointer to function that knows how to delete each item

bagsimple.c uses strings

namedelete from bagsimple.c knows how to free strings

free bagnode_t

Loop over each item

Call delete function passed as parameter (namedelete in bagsimple.c) on each item

Note: this is not the bagnode, it's the item the bagnode holds

Finally free the bag

bagsimple.c uses declarations in bag.h and readline.p to store names in a bag

bagsimple.c

```
#include <stdio.h>
#include <stdlib.h>
#include "bag.h"
#include "readlinep.h"

void nameprint(FILE *fp, void *item);
void namedelete(void *item);

int main() {
        // create a bag
        bag_t *bag = bag_new(); // the bag

        // insert into the bag
        while (!feof(stdin)) {
                char *name = readlinep();
                if (name != NULL) {
                        bag_insert(bag, name);
                }
        }

        // print out bag items
        bag_print(bag, stdout, nameprint);

        // delete the bag
        bag_delete(bag, namedelete);

        return 0;
}
```

By including bag.h and readline.p we get declaration for their functions

Declare functions we will use for this specific use of the bag (this bag stores names as strings)

bag_new creates a new bag

Read a line from stdin and add a new item to bag using bag_insert

When done inputting names, print all elements in bag by calling bag_print

But, a bag that holds items of any type doesn't know how to print the items in the bag (toString in Java tells how to print objects)

Pass a pointer to a print function that knows how to print names

Delete all items in bag after printing

Pass function that knows how to delete (free) items

Bag doesn't know what kind of items it will hold, needs to know how to print and free items

bagsimple.c

```
// print a name
void nameprint(FILE *fp, void *item) {
        char *name = item;
        if (name == NULL) {
                fprintf(fp, "(null)");
        }
        else {
                fprintf(fp, "%s", name);
        }
}

// delete a name
void namedelete(void *item)
{
        if (item != NULL) {
                free(item);
        }
}
```

`nameprint` knows how to print an item of the type stored in this type of bag (here strings)

A pointer to this function is passed to bag_print

`mamedelete` knows how to delete (free) items stored in this bag (strings here)

Agenda