Algorithms for GIS

Programming in C:
Pointers, header files and multiple files

Laura Toma

Bowdoin College

Outline

- Programming in C
  - Pointers
  - .h and .c files
  - Compiling
  - Working with multiple files
  - Using Makefiles

- Programming exercise

Pointers

T x;

- Any variable is stored somewhere in memory and thus has an address, which can be retrieved with operator &

  &x gives the address of variable x

- An address is called a pointer
- The address of a variable of type T is considered to have type T*

  &x has type T*;

- Given an address, we might want to know what is stored at that address. That's called dereferencing the pointer, and it is done with operator *

  T* p;

  *p is of type T, and it is the value stored at address p

- Caveat: Dereferencing an invalid address is a **BUG**.
- A bug of this type doesn't always manifest, and does not manifest in the same way. That is, it might give you a segfault. Or not. Still, your program has a bug in it and its behavior is unpredictable.

Pointers

- Rule: make sure a pointer is valid before you dereference it
  - by assigning it the address of a variable
  - by calling malloc()
  - by assigning it the value of another valid pointer

Pointers

- Rule: make sure a pointer is valid before you dereference it
  - by assigning it the address of a variable
  - by calling malloc()
  - by assigning it the value of another valid pointer
- Perhaps this is boring.. Consider this.
  - You WILL get segfaults
  - You will spend a LONG time figuring it out
  - It’s ALWAYS because you break this one rule

Pointers

- Rule: make sure a pointer is valid before you dereference it
  - by assigning it the address of a variable
  - by calling malloc()
  - by assigning it the value of another valid pointer

- Perhaps this is boring.. Consider this.
  - You WILL get segfaults
  - You will spend a LONG time figuring it out
  - It’s ALWAYS because you break this one rule

- And remember,
  - Bad memory references do not always manifest
  - The program might work fine on one computer, but not on other.

Exercise

- We want to write a function to allocate an array of n element of type T
- We'll write it two ways:
  1. return the array
  2. take the array as parameter

Do both work?

```
//assume T is  a type

T* create(int n) {
    T* result;
    result = (T*)malloc(n*sizeof(T));
    assert(result);
    return result;
}



int n=100;
T* x;
x = create(n);
//is x an array of 100 elements?
```

```
//assume T is  a type

void create(int n, T* a) {
    a = (T*) malloc(n*sizeof(T));
    assert(a);
}



int n=100;
T* x;
create(n, x);
//is x an array of 100 elements?
```

Why doesn't this work?

```
//assume T is  a type


void create(int n, T* a) {
    a = (T*) malloc(n*sizeof(T));
    assert(a);
}



int n=100;
T* x;
create(n, x);
//is x an array of 100 elements?
```

Why doesn't this work?

- a is set correctly inside create()
- But, it's value does not change outside the function

```
//assume T is  a type

void create(int n, T* a) {
    a = (T*) malloc(n*sizeof(T));
    assert(a);
}

int n=100;
T* x;
create(n, x);
//is x an array of 100 elements?
```

Exercises

- Implement this and get a feel for how this bug manifests. Can you find some instances where the program runs seemingly well? What does this show? Can you make it crash?
- Fix it!

C programming

- C gives a lot of freedom for bad style
  - Debugging can be hell. Really.

- Good practices
  - Modularize
  - Separate interface from implementation
  - Program with asserts.
  - Unit testing: Write test modules for EVERYTHING
  - Structure your code assuming there are tests for everything
    - This will change how you design your code

Header files

Header files

Example: implement a linked list

- **list.h**:
  - is the interface to the outside world
  - contains type definitions and signature of functions that are meant to be used by other modules
- **list.c**:
  - implements all functions in list.h

```
/* list.h */


typedef struct node_t {
     int data;
     struct node_t* next;
}

typedef struct list_t {
     Node* head;
} List;


List* init();
…
```

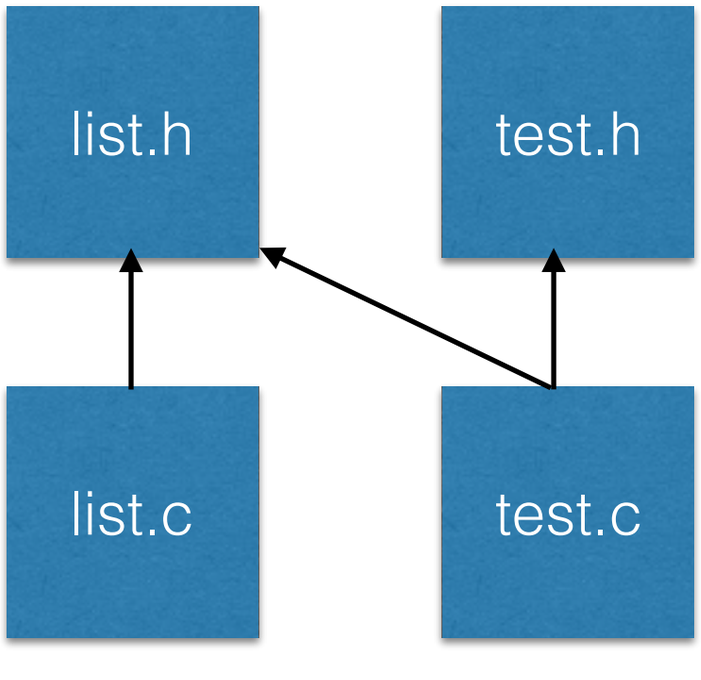
list.h

list.c

```
/* list.c */


#include “list.h”

List* init() {
   //implement init
   …
}
```

Working with multiple files

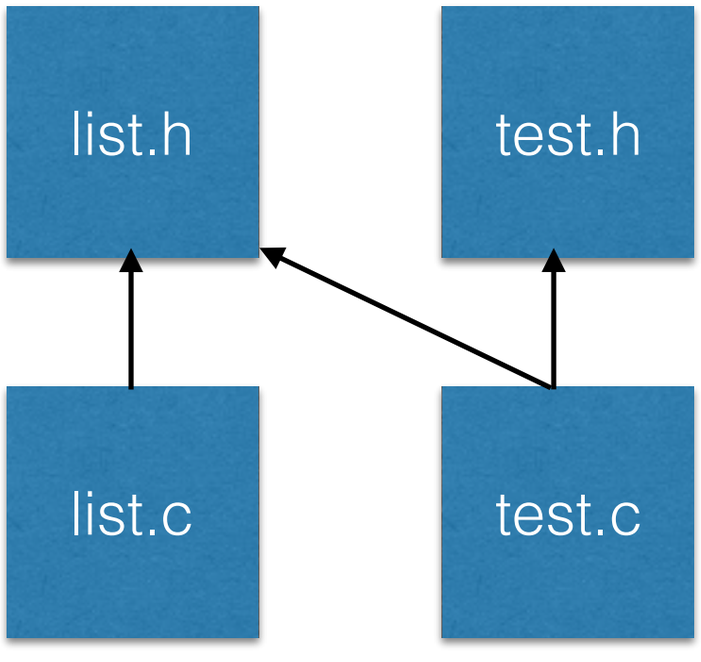
Working with multiple files




```
/* test.c */

#include “test.h”
#include “list.h”

…

int main() {
  ….
}
```

- If test.c needs to use some list functions
  - `#include “list.h”`

Working with multiple files

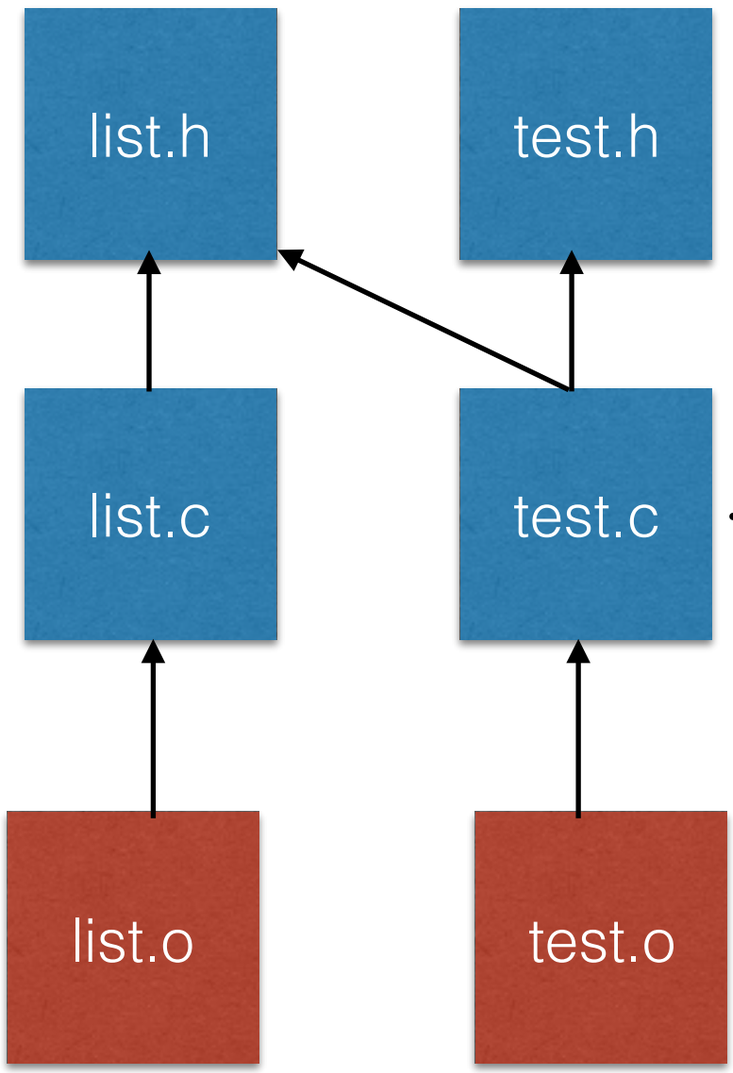



```
/* test.c */

#include “test.h”
#include “list.h”

…

int main() {
  ….
}
```

```
gcc -c list.c -o list.o
gcc -c test.c -o test.o
gcc list.o test.o -o test
./test
```

- Compilation has 2 phases
  - compile only (gcc -c): each xxx.c file ==> xxx.o file
  - for each file that contains a main():
    - link the .o files of the headers that it needs to create the executable

Working with multiple files

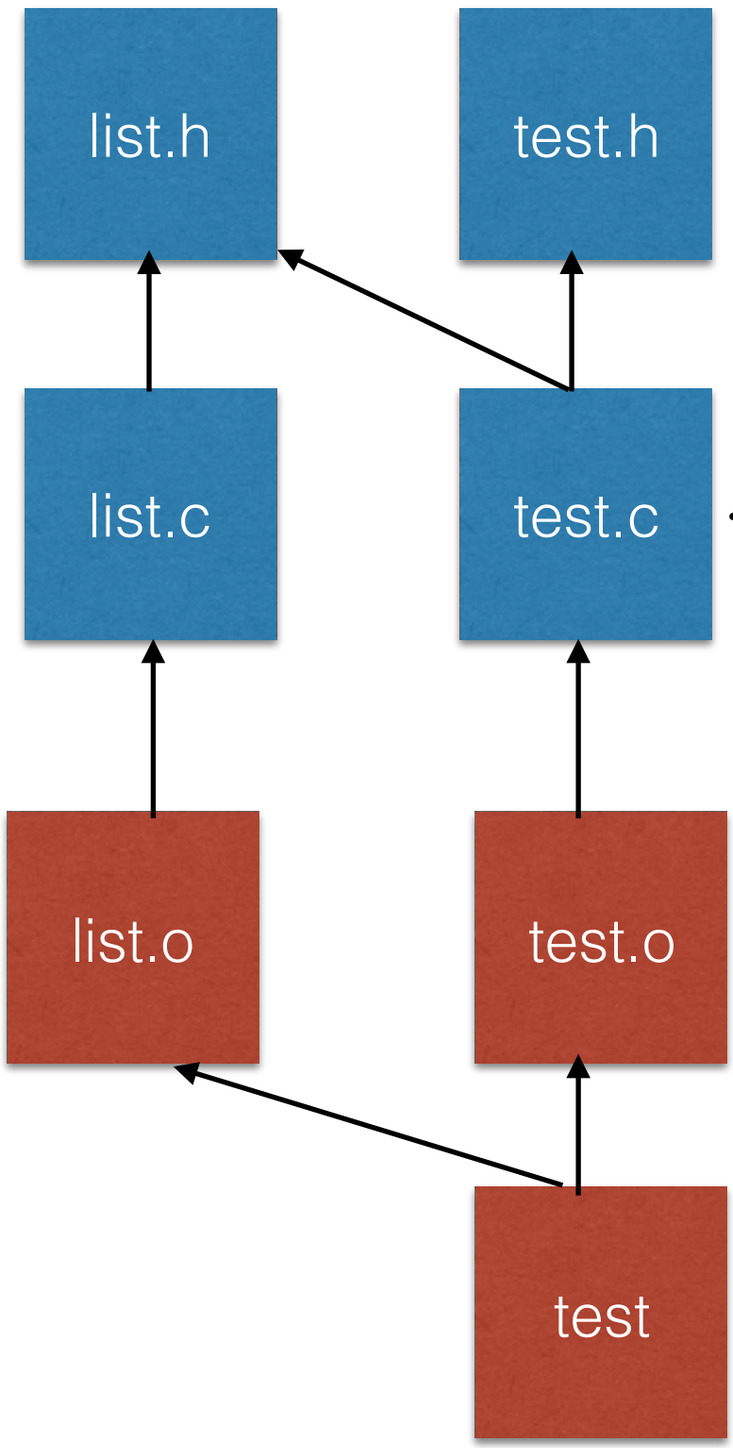



```
/* test.c */

#include “test.h”
#include “list.h”

…

int main() {
  ….
}
```

```
gcc -c list.c -o list.o
gcc -c test.c -o test.o
gcc list.o test.o -o test
./test
```

test.o has a symbol table with external functions, but no info on where they are defined

Working with multiple files



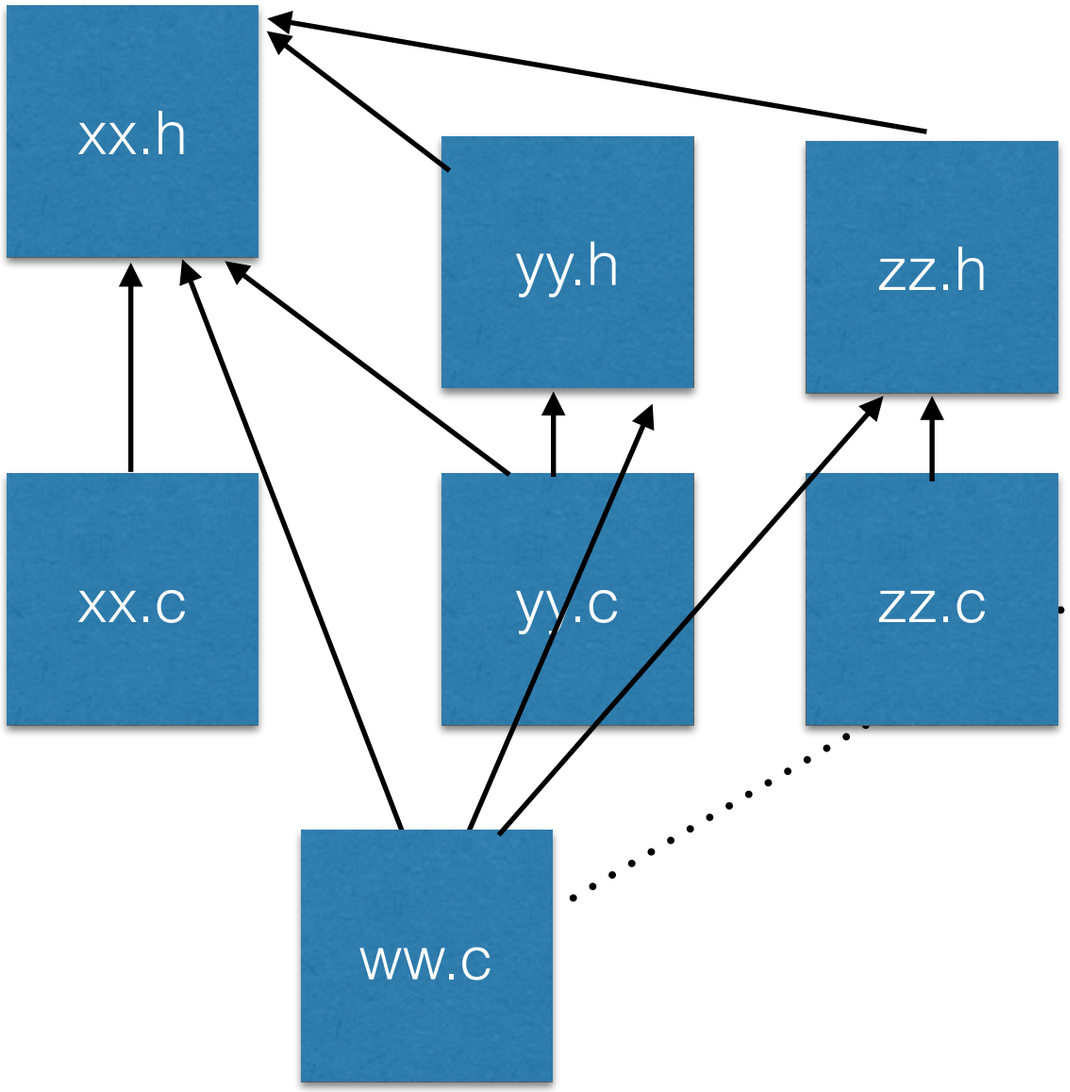

```
/* test.c */

#include “test.h”
#include “list.h”

…

int main() {
  ….
}
```

```
gcc -c list.c -o list.o
gcc -c test.c -o test.o
gcc list.o test.o -o test
./test
```

test.o has a symbol table with external functions, but no info on where they are defined

link phase links the files together and fills in the addresses of functions in the symbol table

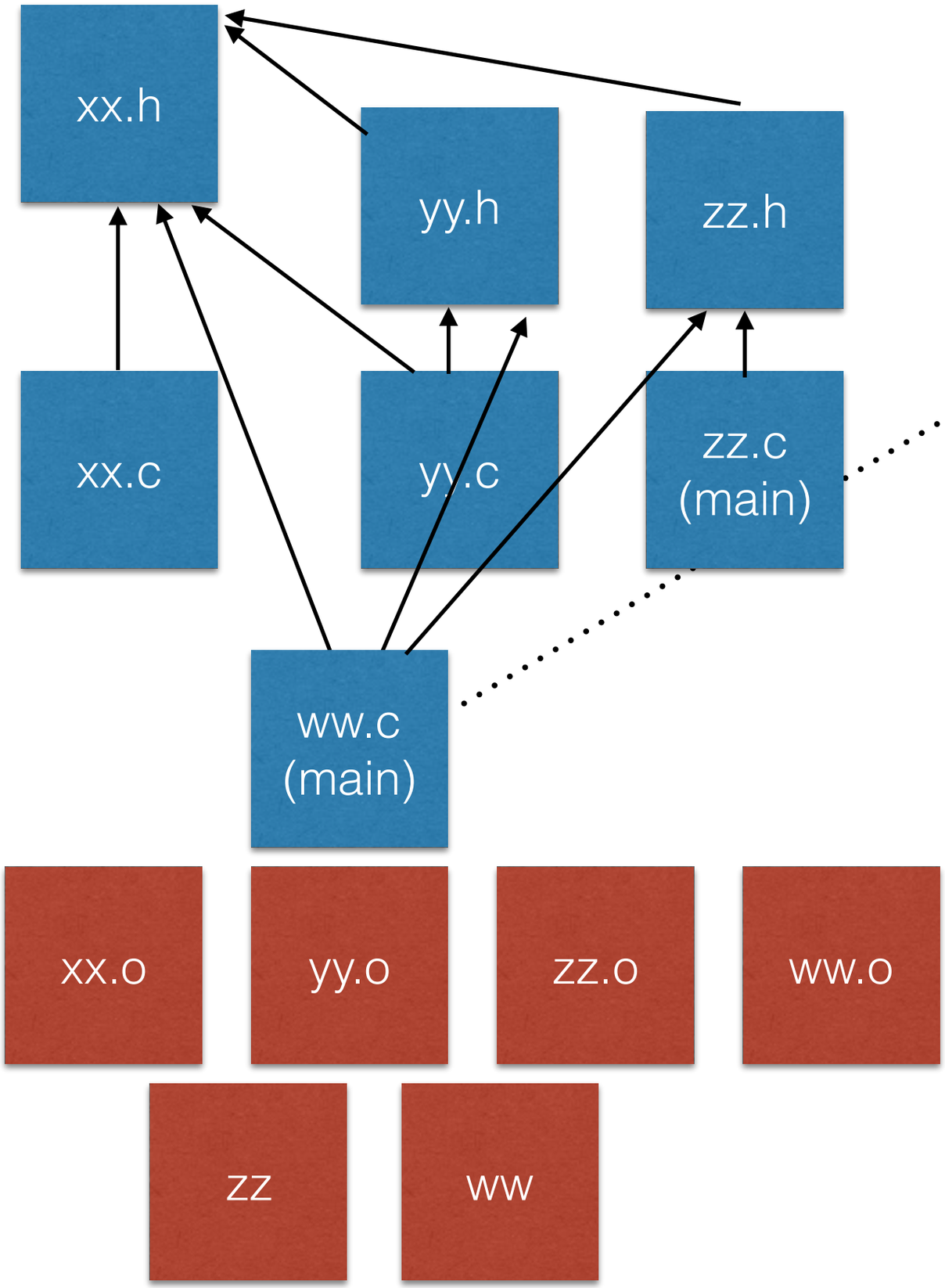
Complex dependency graph




```
/* ww.c */

#include “xx.h”
#include “yy.h”
#include “zz.h”

…
```

- Each file must include all headers it needs
- The dependency graph cannot have cycles
  - If it has cycles ==> very weird compile errors

Complex dependency graph




```
/* ww.c */

#include “xx.h”
#include “yy.h”
#include “zz.h”

…
```

Why?

- For efficiency
  - compiling large projects is slow
  - if change one line in a file, you re-compile only the object files and executables that depend on it, directly or indirectly

- make utility
  - Makefile specifies dependencies
  - ‘make’ keeps track of when files were last modified ==> figures out what changed and what needs to be recompiled