Lecture #07 - Multi-file programs and Make

AMath 483/583

Announcements

- Quick tutorial: using Jupyter to create PDFs
- Homework #2 - online after lecture
- Today:
  - multi-file C programs
  - Python <—> C interfaces
  - Primary References on syllabus

Function Prototypes

```
#include <stdio.h>

int main() {
  int a = 2;
  int b = square(a);  // equals 4
}

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

Function Prototypes

```
#include <stdio.h>

int main() {
  int a = 2;
  int b = square(a);
}

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

Compile Error!

- Compile-time checking for function existence **from top down**
- square(int) is undefined at time of call in main()
- Solution: use function prototype

Function Prototypes

```
#include <stdio.h>

int square(int x);  // “prototype”

int main() {
  int a = 2;
  int b = square(a);  // equals 4
}

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

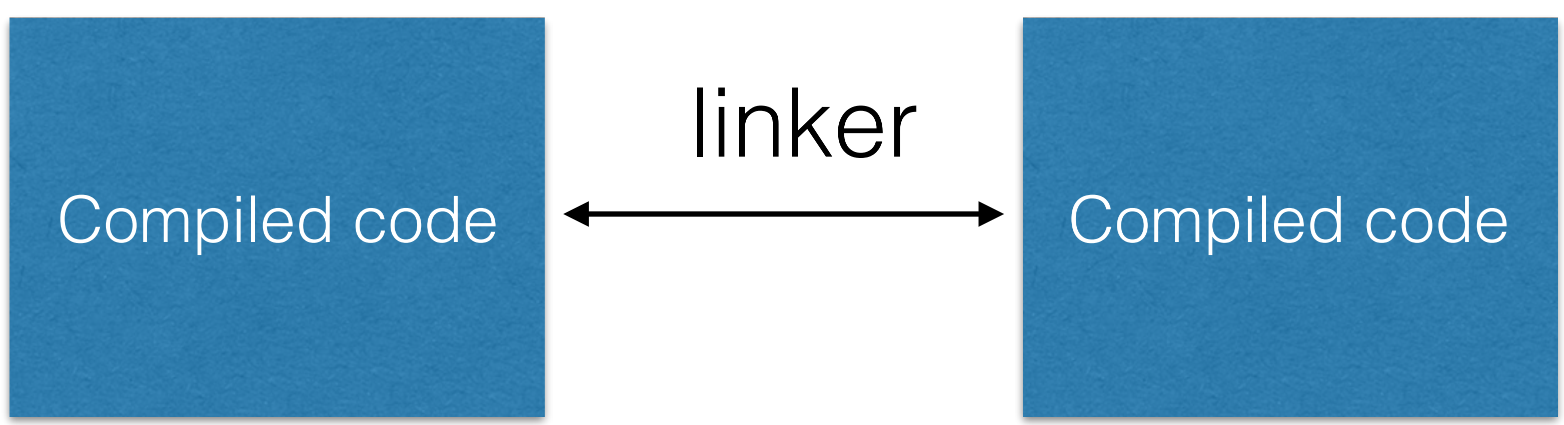
Function Prototypes

```
#include <stdio.h>

int square(int x);  //

int main() {
  int a = 2;
  int b = square(a);
}

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

Also declares the "signature" of `square` - return type and arg type

"Header files" (e.g. "`stdio.h`") contain function prototypes

Compiling and Linking

- 1) compile C code into machine code
- 2) “link” to other compiled code to find missing function definitions
    - “headers” define prototypes
    - “compiled code” contains function definitions
    - 

Two Types of Compiled Code

- **Executables**
  - entry point for a program
  - something with `int main()` defined
- **Libraries**
  - stand-alone, non-executable
  - grouping of functionality
  - distributable

Multi-file Code

```
// main.c
void sub1();
void sub2();
int main() {
  printf(“In main\n”);
  sub1();
  sub2();
  return 0;
}
```

```
// sub1.c
void sub1() {
  printf(“In sub1\n);
}
```

```
// sub2.c
void sub2() {
  printf(“In sub2\n);
}
```

Multi-File Code

- Method 1: compile all three together and link to single executable

```
$ gcc main.c sub1.c sub2.c -o main.exe
```

- Run:

```
$ ./main.exe
In main
In sub1
In sub2
```

Multi-File Code

- Method 2: split into separate “binaries”…

```
$ gcc -c main.c sub1.c sub2.c
$ ls ./*.o
main.o  sub1.o  sub2.o
```

- …explicitly link files:

```
$ gcc ./*.o -o ./main2.exe
$ ./main2.exe
…
```

Multi-File Code

- **Advantage**: if we change `sub2.c` we only need to recompile it (and re-link everything)

  ```
  $ gcc -c sub2.c
  $ gcc ./*.o -o main3.exe
  $ ./main3.exe
  …
  ```

- Large codes have long compile times

Header Files

- Recall at top of main.c:

```
sub1();  // function prototypes
sub2();
```

- How do you know the signature from a different source file? What if you only have a binary?

- **Header files**: *prototypes for corresponding definitions in a binary*

Header Files

```c
// main.c

#include “sub1.h”
#include “sub2.h”

int main() {
  printf(“In main\n”);
  sub1();
  sub2();
  return 0;
}
```

```c
// sub1.h
void sub1();
```

```c
// sub1.c
void sub1() {
  printf(“In sub1\n);
}
```

```c
// sub2.h
void sub2();
```

```c
// sub2.c
void sub2() {
  printf(“In sub2\n);
}
```

Header Files

```
// main.c

#include “sub1.h”
#include “sub2.h”

int main() {
  printf(“In main\n”);
  sub1();
  sub2();
  return 0;
}
```

```
// sub1.h
void sub1();
```

`#include` copies the text in `sub1.h` verbatim before compiling

“C preprocessor”

```
// sub2.c
void sub2() {
  printf(“In sub2\n);
}
```

Header Files

```
// main.c

#include "sub1.h"
#include "sub2.h"

int main() {
  printf("In main\n");
  sub1();
  sub2();
  return 0;
}
```

At compile-time `main.c` is identical to:

```
// main.c

void sub1();
void sub2();

int main() {
  printf("In main\n");
  sub1();
  sub2();
  return 0;
}
```

);

);

Header Files

- Need to tell compiler where to look for headers: happens at compile-time

```
$ gcc -I. -c main.c sub1.c sub2.c
$ gcc ./*.o -o main4.exe
$ ./main4.exe
…
```

Libraries

- `sub1.c` and `sub2.c` contain useful code — distribute as a "library"

  "*implementation of behavior with well-defined interface*"

- **Examples**:
  - blas / lapack — linear algebra libraries
  - openmp / mpi — parallel code libraries

Libraries

- The “don’t do this way”:
  - distribute each `sub.o` file and corresponding header `sub.h`
  - messy
- **Instead**: group object files into one library.

Static Libraries

- Use the “archiver tool”:

```
$ gcc -c sub1.c sub2.c
$ ar rcs sub1.o sub2.o -o libsubs.a
```

- Linking:

```
$ gcc -L. -lsubs main.c -o main5.exe
```

Static Libraries

- Use th

$ gcc
$ ar   a

-Ldir

add **dir** to the list of directories to check for compiled libraries

- Linking

```
$ gcc -L. -lsubs main.c -o main5.exe
```

Static Libraries

- Use th

```
$ gcc
$ ar                                          a
```

-lname

look for libraries matching the name

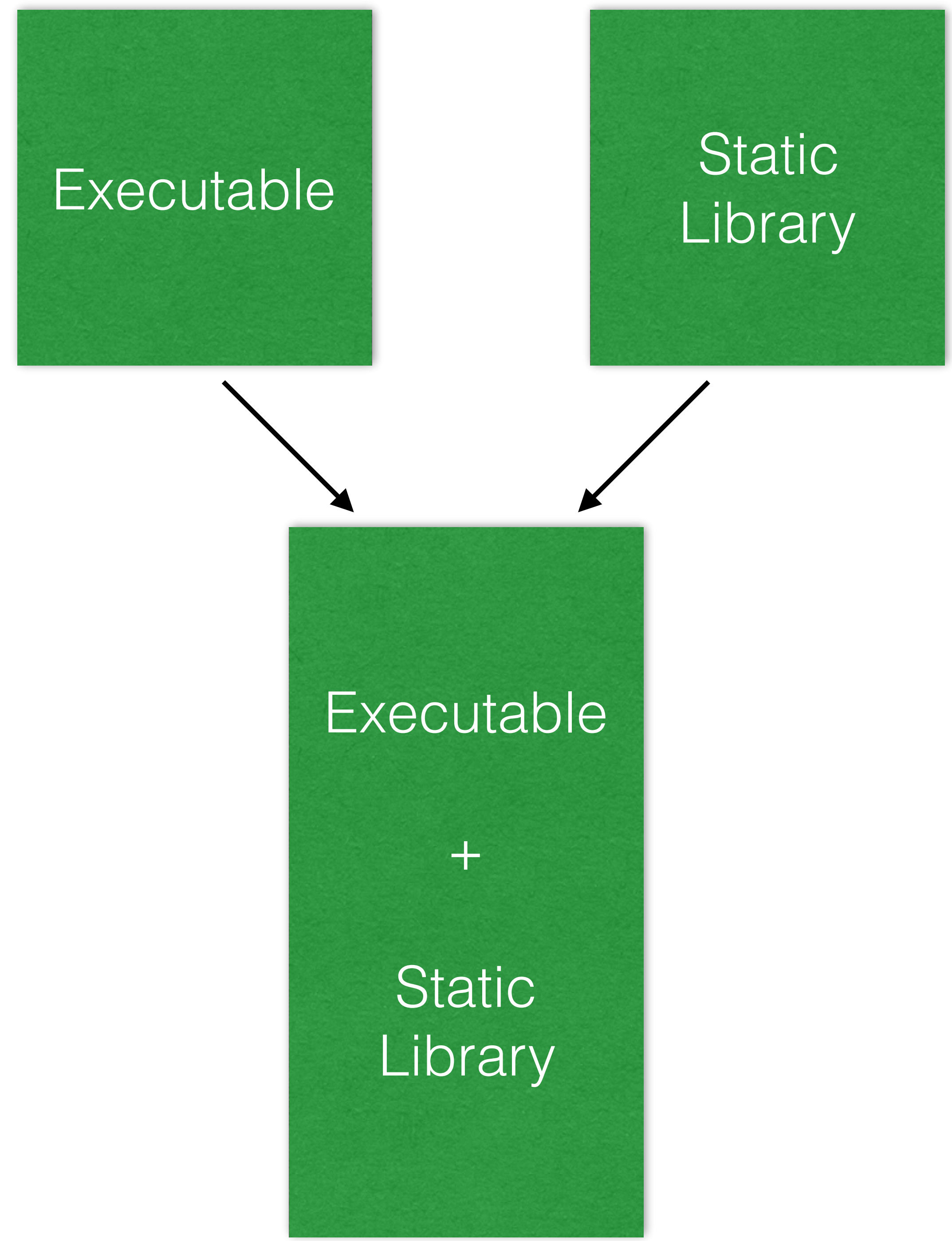
libname.{a,so}

- Linking

```
$ gcc -L. -lsubs main.c -o main5.exe
```

Static Libraries

- Disadvantages:
  - library is loaded at compile time
  - creates a “fat binary” with all library data
- Why make two copies of the library?



Dynamic Libraries

- Use the `-fPIC` and `-shared` flags:

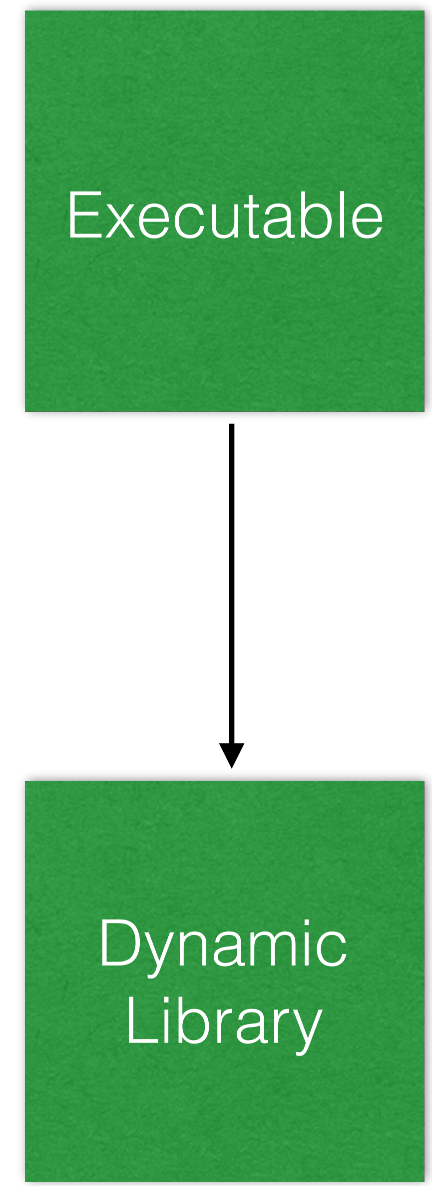
```
$ gcc -c -fPIC sub1.c sub2.c
$ gcc -shared -o libsubs.so ./subs*.o
```

- Linking:

```
$ gcc -L. -lsubs main.c -o main5.exe
```

Dynamic Libraries

- Advantages:
  - library is loaded at run time (linking establishes connection)
  - no copies of contents



Makefiles

- Tedious having to:
  - type compile and linking commands
  - keep track of changed files (no need to recompile unchanged files)
- *Makefiles* are a “compile script”:
  - `make`, `cmake`, `scons`, …

Makefiles

```
main.exe: libsubs.so
	gcc -L. -lsubs main.c -o main.exe

libsubs.so: sub1.o sub2.o
	gcc -shared -o libsubs.so sub1.o sub2.o

main.o: main.c
	gcc -c main.c
sub1.o: sub1.c
	gcc -c -fPIC sub1.c
sub2.o: sub2.c
	gcc -c -fPIC sub2.c
```

Makefiles

```
main.exe: libsubs.so
    gcc -L. -lsubs main.c -o main.exe

libsubs.so: sub1.o sub2.o
    gcc -shared -o libsubs.so sub1.o s

m        ain.c
        c mair
s        ub1.c
    gcc -c -fPIC sub1.c
sub2.o: sub2.c
    gcc -c -fPIC sub2.c
```

target

dependencies

command(s)

Makefiles

- See primary references for details
- You won't need to write them for this class but it is important to understand (used everywhere)

Interfacing Python with C

- C Code can be called by Python
- Often, Python "wrappers" are created:
  - C code: `int is_prime(int n)`
  - Python:

    ```
    def is_prime(n): # calls C's is_prime
    ```
- Software design: easy to use **and** fast

Interfacing Python with C

- Two common tools:
  - **ctypes** — included with Python, simple
  - **cython** — almost different language but very powerful, highly recommended (but beyond scope of class)

Demo

Interfacing with C Code using ctypes