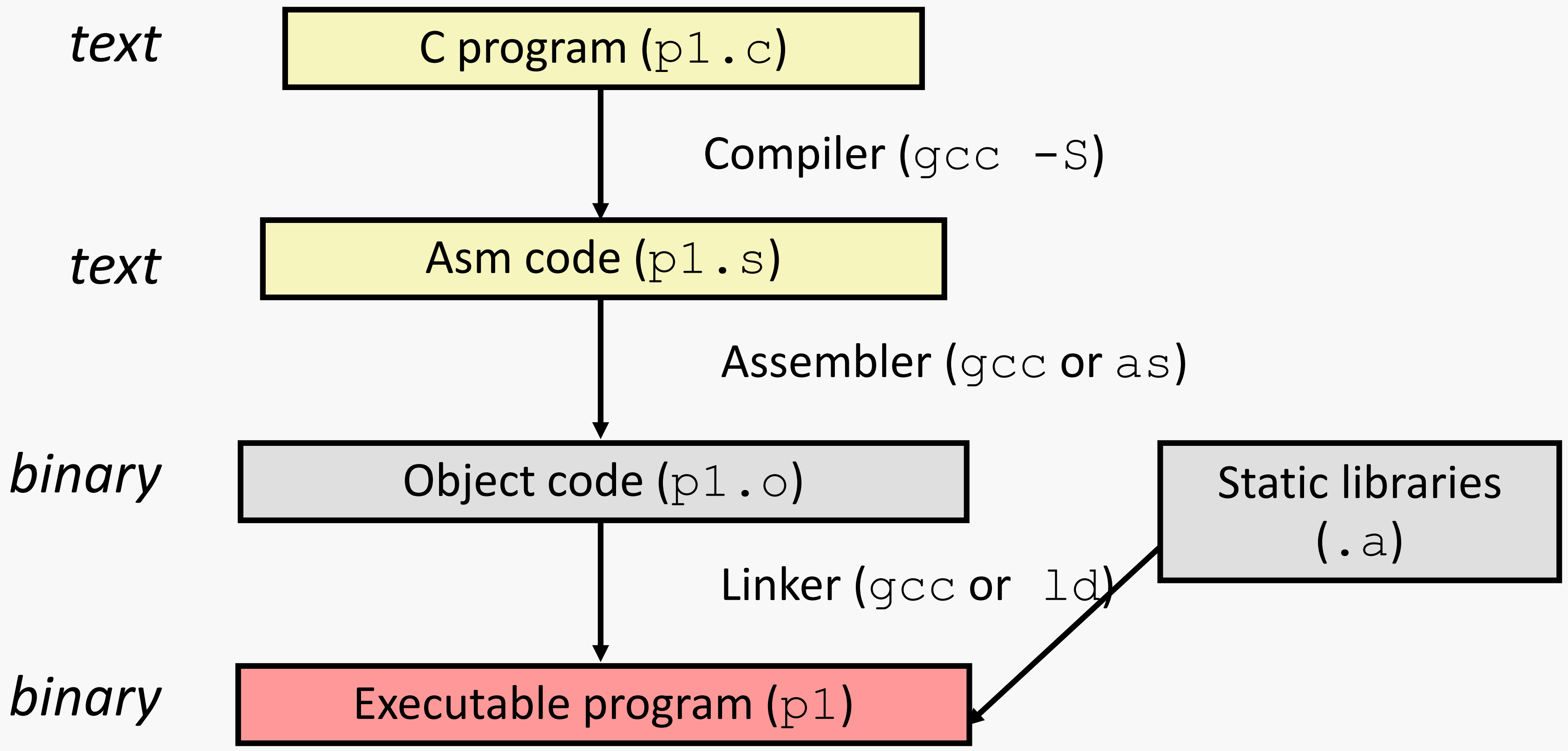
Program Translation



C Source to Assembly Code

C program (`p1.c`)

```
gcc -S -O0 -Wall -m64 p1.c
```

Asm code(`p1.s`)

```
// p1.c
int main() {

   int x, y, t;

   x = 5;
   y = 16;
   t = x + y;

   return 0;
}
```

```
        .file   "p1.c"
        .text
        .globl main
        .type main, @function
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    $5, -4(%rbp)
        movl    $16, -8(%rbp)
        movl    -8(%rbp), %eax
        movl    -4(%rbp), %edx
        addl    %edx, %eax
        movl    %eax, -12(%rbp)
        movl    $0, %eax
        leave
        ret
        . . .
```

Assembly Code to Object Code

Asm code (`p1.s`)

```
as -64 -o p1.o p1.s
```

Object code (`p1.o`)

```
        .file   "p1.c"
        .text
        .globl main
        .type main, @function
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    $5, -4(%rbp)
        movl    $16, -8(%rbp)
        movl    -8(%rbp), %eax
        movl    -4(%rbp), %edx
        addl    %edx, %eax
        movl    %eax, -12(%rbp)
        movl    $0, %eax
        leave
        ret
        . . .
```

```
55
89 e5
83 ec 10
c7 45 fc 05 00 00 00
c7 45 f8 10 00 00 00
8b 45 f8
8b 55 fc
8d 04 02
89 45 f4
b8 00 00 00 00
c9
c3
```

Object Code to Executable

x86 object (`p1.o`)

```
gcc -o p1 -m64 p1.o
```

Executable program (`p1`)

```
55
89 e5
83 ec 10
c7 45 fc 05 00 00 00
c7 45 f8 10 00 00 00
8b 45 f8
8b 55 fc
8d 04 02
89 45 f4
b8 00 00 00 00
c9
c3
```

```
55
89 e5
53
83 ec 04
e8 00 00 00 00
5b
81 c3 74 1d 00 00
8b 93 fc ff ff ff
85 d2
74 05
e8 1e 00 00 00
e8 d5 00 00 00
e8 90 01 00 00
58
5b
c9
c3
. . .
```

The Preprocessor

When a C compiler is invoked, the first thing that happens is that the code is parsed and modified by a *preprocessor*.

The preprocessor handles a collection of commands (commonly called *directives*), which are denoted by the character '#'.

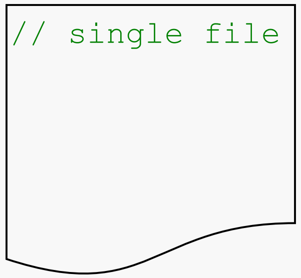
`#include` directives specify an external file (for now a C library file); the preprocessor essentially copies the contents of the specified file in place of the directive.

We will see more interesting preprocessor directives later.

Contents of file `stdio.h` are copied here.

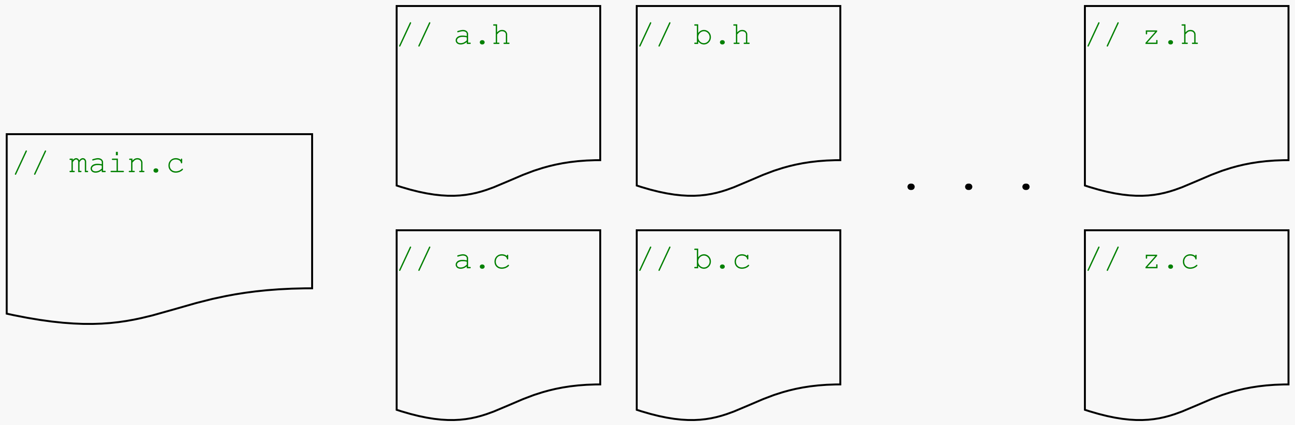
```
#include <stdio.h>
. . .

int main() {

   printf("Hello, world!\n");

   return 0;
}
```

Typical C Program Organization



For very small programs, code is often organized in a single source file; the most common convention uses the extension `c` for C source files.

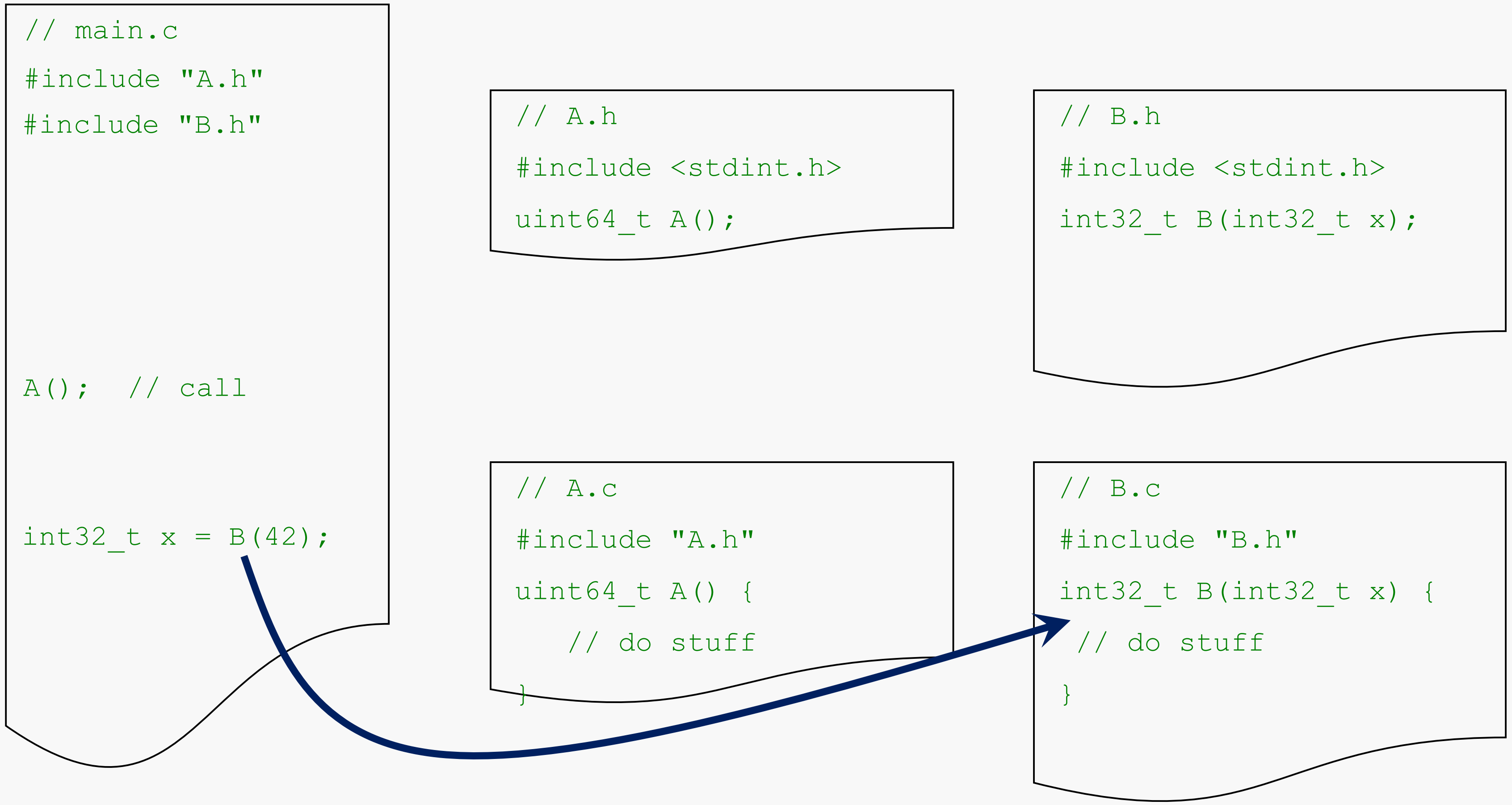
For more interesting C programs, the code is typically organized into a collection of header files (extension `h`) and source files. In most cases, the header files contain only type declarations and function prototypes, while the `c` files contain the corresponding implementations.



`#include` directives are used within both `h` and `c` files to “import” declarations as necessary.

Using Inclusion

#include directives are used within both h and cpp files to "import" declarations as necessary:



```
// main.c
#include "A.h"
#include "B.h"

A();  // call

int32_t x = B(42);
```

```
// A.h
#include <stdint.h>
uint64_t A();
```

```
// B.h
#include <stdint.h>
int32_t B(int32_t x);
```

```
// A.c
#include "A.h"
uint64_t A() {
   // do stuff
}
```

```
// B.c
#include "B.h"
int32_t B(int32_t x) {
  // do stuff
}
```

Controlling Access

Often, a `.c` file includes some functions (and other things) that are needed only in that file:

```
// B.c
#include "B.h"

static void Helper();

int32_t B(int32_t x)
{
 // do stuff
 Helper();
}


void Helper() {
   ...
}
```

Here, `B()` calls `Helper()`, but no other functions outside of the file `B.c` do so…

So, we put the declaration of `Helper()` inside `B.c`, before any calls to it… and make it `static`…

… and we do NOT declare `Helper()` in `B.h`:

```
// B.h
int32_t B(int32_t x);
```

The Multiple Inclusion Problem

It is possible to create some unfortunate situations:

```
// main.c
#include "A.h"
#include "B.h"
```

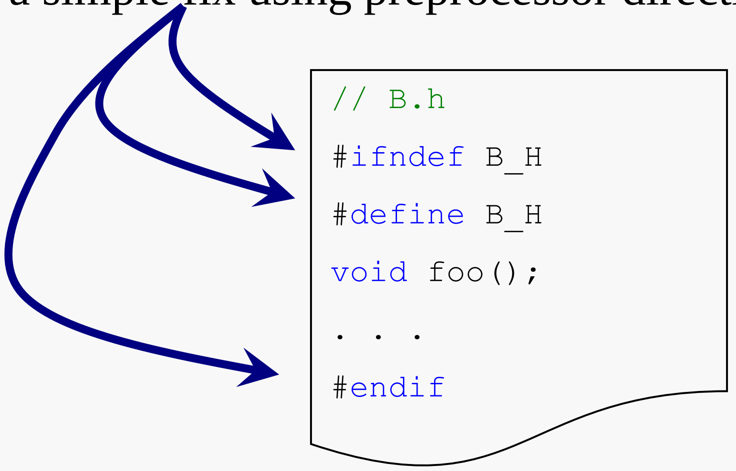
Here, the preprocessor will copy the text of `B.h` into `main.c` twice, once due to the inclusion in `A.h` and once due to the explicit inclusion in `main.c`.

```
// A.h
#include "B.h"
```

This scenario is difficult to avoid in large projects involving many individuals or teams.

```
// B.h
void foo();
```

However, there is a simple fix using preprocessor directives:



```
// B.h
#ifndef B_H
#define B_H
void foo();
. . .
#endif
```

Example: Fibonacci

Here's a short program with a test driver, a Fibonacci function in a second file, and a header file containing the declarations needed for the Fibonacci function:

```
#include <stdio.h>
#include <inttypes.h>
#include "rFibonacci.h"

int main() {

   for (int i = 0; i < 50; i++) {

      printf("Fibo(%d) = %"PRIu64"\n", i, rFibonacci(i));
   }

   return 0;
}
```

```
#ifndef RFIBONACCI_H
#define RFIBONACCI_H
#include <stdint.h>

uint64_t rFibonacci(uint64_t n);

#endif
```

```
#include "rFibonacci.h"

uint64_t rFibonacci(uint64_t n) {
   if ( n < 2 )
      return 1;
   return rFibonacci(n - 1) +
          rFibonacci(n - 2);
}
```

Compilation

We can compile either c file without the other (but not produce an executable):

```
CentOS> gcc -o driver -std=c99 -Wall -W driver.c rFibonacci.c
CentOS> ls -l
total 24
-rwxrw-r--. 1 wmcquain comporg 8578 Sep 25 10:21 driver
-rw-rw-r--. 1 wmcquain comporg  216 Sep 25 10:19 driver.c
-rw-rw-r--. 1 wmcquain comporg  154 Sep 25 10:20 rFibonacci.c
-rw-rw-r--. 1 wmcquain comporg  104 Sep 25 10:20 rFibonacci.h
1029 wmcquain in ~/2505/notes/T08/v1>
```

```
#include <stdio.h>
#include <inttypes.h>
#include "rFibonacci.h"

int main() {

   for (int i = 0; i < 50; i++) {

      printf("Fibo(%d) = %"PRIu64"\n", i, rFibonacci(i));
   }

   return 0;
}
```

OK: the function call matches this declaration; compiler doesn't need to see function definition.

The compiler notes that this call "binds" to this declaration.

Compilation

-c switch tells gcc to compile and exit

```
CentOS> gcc -c -std=c99 -Wall -W driver.c
CentOS> ls -l
total 16
-rw-rw-r--. 1 wmcquain comporg  216 Sep 25 10:19 driver.c
-rw-rw-r--. 1 wmcquain comporg 1616 Sep 25 10:24 driver.o
-rw-rw-r--. 1 wmcquain comporg  154 Sep 25 10:20 rFibonacci.c
-rw-rw-r--. 1 wmcquain comporg  104 Sep 25 10:20 rFibonacci.h
```

If compilation is successful, an object file is created; this is a partial translation of the C source file into machine code.

Linking

The final step in producing an executable is to invoke the linker.

The linker resolves the "notes" left by the compiler for external references (like the function name noted earlier), and writes the final executable file.

With `gcc`, the simplest way to do this is to just invoke `gcc` on the object files…

Linking

```
CentOS> ls -l
total 20
-rw-rw-r--. 1 wmcquain comporg  216 Sep 25 10:19 driver.c
-rw-rw-r--. 1 wmcquain comporg 1616 Sep 25 10:24 driver.o
-rw-rw-r--. 1 wmcquain comporg  154 Sep 25 10:20 rFibonacci.c
-rw-rw-r--. 1 wmcquain comporg  104 Sep 25 10:20 rFibonacci.h
-rw-rw-r--. 1 wmcquain comporg 1424 Sep 25 10:26 rFibonacci.o

CentOS> gcc -o fibo -Wall -W driver.o rFibonacci.o

CentOS> ls -l
total 32
-rw-rw-r--. 1 wmcquain comporg  216 Sep 25 10:19 driver.c
-rw-rw-r--. 1 wmcquain comporg 1616 Sep 25 10:24 driver.o
-rwxrw-r--. 1 wmcquain comporg 8578 Sep 25 10:27 fibo
-rw-rw-r--. 1 wmcquain comporg  154 Sep 25 10:20 rFibonacci.c
-rw-rw-r--. 1 wmcquain comporg  104 Sep 25 10:20 rFibonacci.h
-rw-rw-r--. 1 wmcquain comporg 1424 Sep 25 10:26 rFibonacci.o
```

-o switch tells gcc what to call executable

can invoke gcc with object files to link them and produce executable

Compilation Errors

The compiler will generate error messages if it cannot resolve the references it encounters:

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>
//#include "rFibonacci.h"

int main() {

   for (int i = 0; i < 50; i++) {

      printf("Fibo(%d) = %"PRIu64"\n", i, rFibonacci(i));
   . . .
```

Omit this include directive and the compiler cannot find a declaration that matches the call to rFibonacci().

```
CentOS> gcc -c -std=c99 -Wall -W driver.c
driver.c: In function ‘main’:
driver.c:9:7: warning: implicit declaration of function
‘rFibonacci’ [-Wimplicit-function-declaration]
       printf("Fibo(%d) = %"PRIu64"\n", i, rFibonacci(i));
       ^
driver.c:9:7: warning: format ‘%lu’ expects argument of type ‘long
unsigned int’, but argument 3 has type ‘int’ [-Wformat=]
```

Linker Errors

The linker may also produce error messages:

```
#include <stdio.h>
#include <stdint.h>
#include "rFibonacci.h"

int main() {

   for (int i = 0; i < 50; i++) {

      printf("Fibo(%d) = %lld\n", i, rFibonacci(i));
   }

   return 0;
}
```

```
#ifndef RFIBONACCI_H
#define RFIBONACCI_H
#include <stdint.h>

uint64_t rFibonacci(uint64_t n);

#endif
```

OK

```
#include "rFibonacci.h"

uint64_t Fibonacci(uint64_t n) {
   if ( n < 2 )
      return 1;
   return rFibonacci(n - 1) +
          rFibonacci(n - 2);
}
```

Now, linker cannot find a definition (implementation) that matches the function declaration the compiler matched the call to… that's an error.

Linker Errors

Unfortunately, linker error messages are less direct than compiler error messages:

```
CentOS> gcc -o driver -std=c99 -Wall -W driver.c rFibonacci.c
/tmp/ccHhmn0D.o: In function `main':
driver.c:(.text+0x1a): undefined reference to `rFibonacci'
/tmp/ccYcsMDA.o: In function `Fibonacci':
rFibonacci.c:(.text+0x27): undefined reference to `rFibonacci'
rFibonacci.c:(.text+0x3a): undefined reference to `rFibonacci'
collect2: error: ld returned 1 exit status
```

The message does not refer to a specific line of source code.

But, it does name a specific function and it implies there's a difference between the declaration of that function and the definition (implementation) of that function.

Caesar Cypher Redux

The Caesar cypher program we saw earlier can be improved by separating the code that manages the input/output from the code that actually performs the translation:

```
// cypher.c
. . .
#include "caesar.h"

uint32_t processFile(int shiftAmt,
                     const char* const fileName);
. . .
int main(int argc, char** argv) {
   . . .
}
. . .

uint32_t processFile(int shiftAmt,
                     const char* const fileName)
{
   . . .
}
. . .
```

```
// caesar.h
#ifndef CAESAR_H
#define CAESAR_H
#include <stdint.h>

char mapChar(char Original,
                uint8_t shiftAmt);

#endif
```

```
// caesar.c
#include "caesar.h"

. . .
char mapChar(char Original,
                uint8_t shiftAmt) {
   . . .
}
```