Running a C Program

Kurt Schmidt

Dept. of Computer Science, Drexel University

April 26, 2021
Objectives

• Be introduced to the processes that turn C source code into a native executable, and run it
• Use this knowledge to better understand errors
Intro
In an IDE, when you push the "Run" button, a number of things happen:

1. Dirty flags are checked on open editor windows
   - Saves, or prompts to save
2. Source code is run through the **precompiler**
3. The output is then compiled
4. The **linker** makes the various object files into an executable
5. The **loader** reads your program from disk and runs it
   - Or, the **debugger** performs a similar task
For this lecture I will use the Gnu C Compiler

gcc (Ubuntu 7.4.0-1ubuntu1 18.04.1) 7.4.0

• As previously noted, we can compile a C program like this:
  
gcc f1.c f2.c ... fn.c -o exe_name

• gcc is actually a wrapper

• gcc either performs these functions, or calls another program to do so:
  1. Precompile
  2. Compile
  3. Assemble
  4. Link

• Note, gcc (like most compilers) is not strictly ISO/ANSI
Notes on Compilers

- We have gcc, tcc, and clang loaded on tux
- Can be used interchangeably (for these examples)
  - Mostly
  - None are strictly ANSI compliant
  - tcc and clang approach compliance
  - Error messages will differ
Precompiler
Precompiler (Preprocessor)

- Removes comments
- Evaluates the preprocessor directives
  - Statements that being w/a #:
    - #include #if #ifdef #ifndef #define #endif #pragma etc.
  - #included code is processed recursively

```c
#include
```

- Don’t compile header files (.h) directly
- #include header files where needed
- Do not #include source files (.c, etc.)
Precompiler (cont.)

- Input is source code (.c)
- Output is also text
  - Actually, preprocessing is not usually done separately
  - However, it is useful to think of it as a separate pass
- To see example output (to stdout, throw the -E flag:

  $ gcc -E hello.c | more  # ignore $  

- This stops processing before the compiling stage
Compiler
Very generally, a compiler translates from one language to another.

Usually translates from a higher-level language to a lower one

(See decompiler; it's also a compiler)

The C compiler translates from (processed) C source to object code

- *Object code* is machine code, but *not* an executable program (yet)

\[1\] In two steps, maybe
Assembler

- Traditionally, C compilers compile to Assembly code (.asm)
- This code would then be assembled into object code
- gcc still does this
- Other compilers (such as Tiny C) go right to object code
- This distinction is not so helpful for you to understand right now
- To compile only to the assembler code:
  
  $ gcc -S hello.c  # ignore $
  
  - Yields a file called hello.s
Producing Object Files

- The compiler can turn individual source (.c) files into object (.o) files
- This is a very handy feature
  - For larger systems, we needn’t re-compile all the code after making a single change
  - We can just re-compile the updated source file, then re-link all the object code into a new executable
- To compile to object code, no linking, throw the (-c) (compile-only) flag:

```
$ gcc -c hello.c  # ignore $
```

- Yields a file called `hello.o`<sup>1</sup>

<sup>1</sup>`hello.obj` on Windows
Linker
The linker produces an executable image from object files (and other libraries)

- It combines the object files into a single file
- It resolves external references
- It identifies the entry point of the program (main)
- It is a separate program (ld)
  - The gcc wrapper passes options to the linker
  - Better than invoking it yourself
External References

External Symbols

- A function might refer to a global variable defined in another file
- A function might call a function defined in another file
- These are *external references*
- The compiler might compile these files in separate compilation units
  - Which is why the compiler wants prototypes, and *extern* declarations
- It is up to the linker, when combining the object files, to resolve these references
Let *gcc* call the linker for you

- The *gcc* wrapper passes options to the linker
- Better than invoking it yourself
- If you pass object files to *gcc*, only linking need occur:

  ```
  $ gcc chocolate.o cookie.o main.o -o myProg     # ignore $
  ```

- *gcc* knows what it’s doing. This works fine:

  ```
  $ gcc source.c processed.i assembly.s object.o -o zoo     # $
Multiple-File Compiles
Subsequent frames refer to these files:

```c
char *projName = "Foo" ; /* defined and initialised here */
int foo( int a, int b )
{
    return 3 * a + b ;
}
```

Copy them from the Lectures subdirectory, or create them.
foo.h

```c
#ifndef __KS_FOO_H_
#define __KS_FOO_H_

#include "foo.h"

/* Declared here for all to use */
/* NOT a definition. No memory */
extern char *projName ;

int foo( int ) ; /* prototypes are automatically extern */

#endif /* __KS_FOO_H_ */
```
```c
#include <stdio.h>
#include "foo.h"

int main( int argc, char **argv )
{
    int i, r;
    i = 12;
    r = foo( i );
    printf( "%s: The answer is: %d\n", projName, r );
    return 0;
}
```
The `#include (main.c, lines 1&2)` statement is replaced with the contents of the named file

Maybe. The `#ifndef (foo.h, line 1)` statement prevents the file from being included more than once in the same compilation unit.

System header files are placed in angle brackets (`main.c, line 1`), so the compiler knows where to look for them.

User-defined header files are placed in quotes (`main.c, line 2`), and are expected to be in the current directory.
Function Declarations

- Header files contain *declarations* and *type definitions*
- A *prototype* (`foo.h`, line 10) *declares* the function
  - Allows the compiler to check that you called the function (`main.c` line 9) correctly
  - A function *definition* (`foo.c` lines 5-8) can serve as a declaration, too
- Prototypes are tidier
• Modify line 9 of `main.c`, add another argument:

```c
r = foo( i, 13 );
```

• Try to compile it, note the error:

```bash
$ gcc -c main.c  
```

```
main.c: In function 'main':
main.c:9:6: error: too many arguments to function 'foo'
r = foo( i, 13 );
     ^~~
In file included from main.c:2:0:
foo.h:8:5: note: declared here
int foo( int ); /* prototypes are automatically extern */
     ^~~
```
The Function Definition

- Note the `#include` statement in `foo.c`, line 1
- It is helpful to check that the function definition matches the declaration that all calls are using
  - This check will not be performed at linking \(^1\)
- Consider the modified `foo.c` on the next slide
  - Note the missing `#include` statement
  - Note the change to the number of arguments
  - Compile and run it

\(^1\) C only; C++, which allows overloading of functions, will check the call against the function
Unmatched Call – Example

```c
1 char *projName = "Foo" ; /* defined and initialised here */
2
3 int foo( int a, int b )
4 {
5     return 3 * a + b ;
6 }
```

```
$ gcc main.c foo.c
$ ./a.out
```

```
Foo: The answer is: -700597028
```

- Clearly erroneous
- You’ll get different answers, each time it’s run, potentially
Unmatched Call (cont.)

• Replace the #include statement in foo.c:

```c
#include "foo.h"
```

• Try to compile again:

```bash
$ gcc main.c foo.c
```

```bash
foo.c:5:5: error: conflicting types for 'foo'
int foo( int a, int b )
    ^~~
In file included from foo.c:1:0:
foo.h:11:5: note: previous declaration of 'foo' was here
int foo( int ); /* prototypes are automatically extern */
    ^~~
```

• *Now* this error is caught
The Tiny C Compiler (tcc)
Multi-File Project – Source File

- Small
  - 247 kB vs. gcc’s 987 kB
- Fast
  - Doesn’t use Assembly code as an intermediate step
  - Also seems to generate smaller (faster) code
- Approaches ISO C99 compliance
  - Has many GNU C extensions
- Safe (?)
  - Optional memory and bounds checker
  - (As does gcc and clang, I believe)
- Supports C scripting
  - “Run” C code directly
Compiling with `tcc`

- Works, and used, much like `gcc`
- Can produce an executable directly:
  ```
  $ tcc main.c foo.c -o foo  # $
  ```
- Can produce object files for later linking:
  ```
  $ tcc -c main.c  
  $ tcc -c foo.c  
  $ tcc main.o foo.o -o foo  # $
  ```
Compiling with \texttt{tcc}

- Programs can be run straight from source:
  
  $\texttt{tcc \ -run \ hello.c \ # \ $}$

- Handy invoking from inside editor while developing/debugging

- To make a C program into a script:
  
  - Give execute permissions on the file
  - Provide a sha-bang to the compiler in the first line:
    
    \texttt{#!/usr/bin/tcc \ -run}$

- Note, \texttt{gcc} and others won’t compile it, now