Computer Programming: Skills & Concepts (CP)
Libraries and separate compilation

Julian Bradfield

Monday 20 November 2017

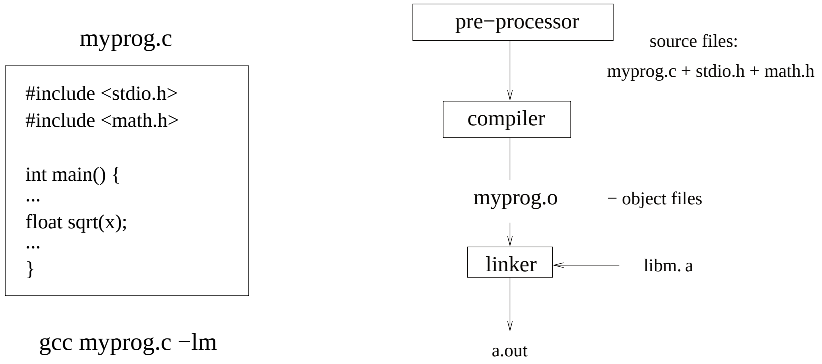
Compiling a C program

Is actually a three stage process. . .

- The ‘C pre-processor’ adds all the #include files and expands the #define statements.
- The ‘C compiler’ compiles the *source* files into *object* files.
- The ‘Linker’ links the object files with libraries into an *executable* that you can run.

```
gcc myprog.c -lm
```

The stages of compilation



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

int main() {
...
float sqrt(x);
...
}
```

Actually, nowadays the ‘link’ stage first checks for the existence of a *shared library* `libm.so`. If it finds one, it notes the fact, but doesn’t link it. Then the library is linked to your program as the first step of running it. *Static libraries* really are brought in at link time.

The pre-processor

```
#include <stdio.h>   /*  These header files get added
#include <stdlib.h>   *  directly into the program code
#include <math.h>     *  by the pre-processor.    */

#define SIZE 20      /* Pre-processor will put 20 everywhere
                      * SIZE appears in code
                      * (except inside quotes) */

int main() {
  int p, q;
  double x[SIZE], y[SIZE];
  ...              /* will get changed to x[20], y[20] */
  for (p=0; p < SIZE; p++)
  ...              /* will get changed to have p < 20 */
}
```

To do compilation only

To compile into an object file, and not link.

```
gcc -c myprog.c
```

A file is produced called `myprog.o`
To link object files:

```
gcc myprog.o -lm
```

executable file `a.out` is produced.
To produce a different name of executable:

```
gcc -o name myprog.o -lm
```

(To run just the pre-processor) **Not** usual to do this manually.

```
cpp myprog.c
```

Some more compiler flags

Optimization:

`-O`: Compile the program for performance.

`-O2`/`-O3`: Aggressive optimisations. At the expense of compile time and memory usage.

```
gcc -O3 myprog.c -lm
```

It is unfortunately not uncommon for high levels of optimisation to have bugs. If you ever have a bug you *really* can't understand, always try compiling without optimisation!
De-bugging:

-g flag adds information to enable a debugger tool to work.

```
gcc -g myprog.c -lm
```

You can combine optimisation and debugging, but optimised code is often very hard to debug.

Functions in separate files

> *A program* `prog1.c` *consists of its* `main` *function, with a single function* `func1()`. *Also the math library is used.*

Place function in a separate file `func1.c`. Compile both:

```
gcc -c prog1.c
gcc -c func1.c
```

Then link together into `a.out`

```
gcc prog1.o func1.o -lm
```

Why?
- function can easily be re-used elsewhere.
- No need to re-compile `func1` if it hasn't changed (good for large files)!

A simple program

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
double func1(double y);

int main() {
  double x,y;
  y = 0.5;
  x = func1(y);
  printf("x was %f\n",x) ;
  return EXIT_SUCCESS;
}

double func1(double y) {
  return sin(y)*cos(y);
}
```

Split into 2 files

Make two files `prog1.c` and `func1.c`.

- `prog1.c` contains just the main body of original program;
- `func1.c` contains just the function `func1`, plus some `#include` statements;
- Must include the following *prototype* at top of `prog1.c`:
  `double func1(double y);`

extern declaration

Indicates to the compiler that a variable or function is to be found in another file – will be resolved later by the linker.

Only applies at global scope. *i.e only to global variables and functions.*

Function prototypes are automatically `extern`. Variables are not, so must write `extern` for external variables:

```
/* This variable is found in another object file */
extern int the_number;
```

Where to put these external declarations?

- Can be messy with many functions in one file.
- We can use the pre-processor.

Header file option

Make three files `prog1.c`, `func1.h`, and `func1.c`.

- `prog1.c` contains the main body of original program:
  + also contains `#include "func1.h"`
  – but no longer has the prototype definition for `func1`.
- `func1.c` contains just the function `func1`, plus some `#include` statements;
- `func1.h` is just the following declaration:
  `double func1(double y);`

Header files

Files containing function declarations are usually called *header files*.

Convention:

- `function1.h` contains function headers.
- `function1.c` contains the functions themselves.

To add functions to your program:

- `#include "function1.h"`
- `gcc myprog.c function1.o`

just as we have been doing with Descartes.

Might be many functions per file.

Compilation (summary)

- Compilation is a three stage process.
- Can compile into object files separately.
- Multiple object files can be linked into a single program.
- Need to declare functions with prototypes.
- Use of header files.

make and Makefiles

`make` is a tool for automating the building of programs.

A `Makefile` consists of a number of rules.
One rule consists of:

- **target**: a target is a file(s) to be built.
- **dependencies**: a list of files that the target relies on.
- **commands**: how to build the target.

`make` ⟨*target_file*⟩; will build the file based on the rules.

A simple Makefile

```
func1.o:     func1.c func1.h project.h
        gcc -c func1.c
# NOTE: 1st char of prev line is TAB (ascii 9), NOT 8 spaces!

func2.o:      func2.c func2.h project.h
        gcc -c func2.c

program:     func1.o func2.o program.c project.h
        gcc -o program program.c func1.o func2.o -lm

all:    program
```

- `project.h` has constants for the whole project. All files depend on it.
- `func1.o` depends on `func1.c` and `func1.h`.
- program depends on `func1` and `func2`.

Makefiles

- Very flexible, powerful – and complicated!
- MACROS – constants that can be defined
- Special macros: `$@` is the name of the file to be made:

  ```
  CFLAGS= -c
  printenv:  printenv.c
          gcc $(CFLAGS) $@.c -o $@
  ```

- Makefiles can call any command, and can be used for a wide variety of tasks.
- make has built-in rules: e.g. for making object files from C files.
- Makefiles are often automatically generated by a higher level project management system.

If your program has more than one file, or uses libraries, use a Makefile!
It saves typing and errors. . .