LinuxFocus article number 229
http://linuxfocus.org

by Lorne Bailey
<sherm_pbody/at/yahoo.com>

About the author:

Lorne lives in Chicago and works as a computer consultant specializing in getting data in and out of Oracle databases. Since making the switch to programming exclusively in a *nix environment, Lorne has completely avoided 'DLL Hell'. He is currently working on his Master's Degree in Computer Science.

GCC - the root of all

Abstract:

This article assumes you know the basics of the C language, and will introduce you to using gcc as a compiler. We will make sure that you can invoke the compiler from the command line on simple C source code. Then we take a quick look at what's actually happening and how you can control the compilation of your programs. We also take a very brief look at using a debugger.

GCC rules

Can you imagine compiling Free software with a closed source, proprietary compiler? How do you know what's going into your executable? There could be any kind of back door or Trojan. Ken Thompson, in one of the great hacks of all time, wrote a compiler that left a back door in the 'login' program and perpetuated the trojan when the compiler realized it was compiling itself. Read his description of this all time classic here. Luckily, we have gcc. Whenever you do a `configure; make; make install` gcc does a lot of heavy lifting behind the scenes. How do we make gcc work for us? We will start writing a card game, but we will only write as much as we need to to demonstrate the compiler's functionality. Since we're starting from scratch, it takes an understanding of the compile process to know what needs to be done to make an executable and in what order. We will look at the general overview of how a C program gets compiled and the options that make gcc do what we want it

to do. The steps (and the tools that do them) are Pre-compile (gcc -E), Compile (gcc), Assembly (as), and Link (ld).

In the Beginning...

First though, we should know how to invoke the compiler in the first place. It's simple, actually. We will start with the all time classic first C program. (Old-timers will have to forgive me).

```
#include <stdio.h>

int main()
{
  printf("Hello World!\n");
}
```

Save this file as `game.c`. You can compile it on the command line by running:

```
gcc game.c
```

By default, the C compiler creates an executable named `a.out`. You can run it by typing:

```
a.out
Hello World
```

Every time you compile a program, the new `a.out` will overwrite the previous program. You will not be able to tell which program created the current `a.out`. We can solve this problem by telling gcc what we want to name the executable with the `-o` switch. We'll call this program `game`, though we could name it anything we want since C doesn't have the naming restrictions that Java does.

```
gcc -o game game.c

game
Hello World
```

At this point, we're pretty far from having a useful program. If you think that's a bad thing, you might want to consider the fact that we have a program that compiles and runs. As we add functionality little by little to this program, we want to make sure that we keep it runnable. It seems that every beginning programmer wants to write 1,000 line of source code and then fix it all at once. No one, I mean no one, can do that. You make a little program that runs,you make changes. and make it run again. This limits the errors you have to correct at one time. Plus, you know exactly what you just did that doesn't work, so you know where to concentrate. This keeps you from creating something that **you** think should work, and maybe even compiles, but can never become an executable. Remember, just because it compiles doesn't mean it's correct.

Our next step is to create a header file for our game. A header file concentrates data types and function declarations in one place. This ensures the data structures are consistently defined so that every part of our program sees everything exactly the same way.

```
#ifndef DECK_H
#define DECK_H

#define DECKSIZE 52

typedef struct deck_t
{
  int card[DECKSIZE];
  /* number of cards used */
  int dealt;
}deck_t;

#endif /* DECK_H */
```

Save this file as `deck.h`. Only `.c` files compile, so we have to change our game.c. On line 2 of game.c, write `#include "deck.h"`. On line 5, write `deck_t deck;` To make sure we didn't break anything, compile it again.

```
gcc -o game game.c
```

No errors, no problem. If it doesn't compile for you, work on it until it does.

Pre-compile

How does the compiler know what a `deck_t` type is? Because during pre-compilation, it actually copies the "deck.h" file into the "game.c" file. The precompiler directives in the source code itself are prefixed by a "#". You can invoke the precompiler through the gcc frontend with the `-E` switch.

```
gcc -E -o game_precompile.txt game.c
wc -l game_precompile.txt
  3199 game_precompile.txt
```

Almost 3,200 lines of output! Most of this comes from the `stdio.h` include file, but if you look at it, our declarations are in there, too. If you don't give an output file name with the `-o` switch, it writes to the console. The process of pre-compilation gives greater flexibility in the code by accomplishing three major objectives.

1. Copies the "#included"d files into the source file to be compiled.
2. Replaces "#define"d text with the actual value.
3. Replaces macros in line whenever they are called.

This allows you to have named constants (i.e. DECKSIZE represents the number of cards in a deck) used throughout the source defined in one place and automatically updated everywhere whenever the value changes. In practice, you almost never use the `-E` switch by itself, but let it pass it's output to the compiler.

Compile

As an intermediate step, gcc translates your code into Assembly language. Do do this, it must figure out what you intended to do by parsing through your code. If you make a syntax error, it will tell you so and the compile will fail. People sometimes mistake this one step for the entire process. But there's a lot more work left for gcc to do.

Assembly

`as` turns the Assembly code into object code. Object code can't actually run on the CPU, but it's pretty close. The compiler option `-c` turn a .c file into an object file with a .o extension. If we run

```
gcc -c game.c
```

we automatically create a file named game.o. Here we have stumbled on an important point. We can take any .c file and create an object file from it. As we see below, we can then combine those object files into an executable in the Link step. Let's go on with our example. Since we're programming a card game and we have defined a deck of cards as a `deck_t`, we will write a function to shuffle the deck. This function takes a pointer to a deck type and loads it with a random set of values for cards. It keeps track of which cards have already been used with the 'drawn' array. This array of DECKSIZE members keeps us from duplicating a card value.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "deck.h"

static time_t seed = 0;

void shuffle(deck_t *pdeck)
{
  /* Keeps track of what numbers have been used */
  int drawn[DECKSIZE] = {0};
  int i;

  /* One time initialization of rand */
  if(0 == seed)
  {
    seed = time(NULL);
    srand(seed);
  }
  for(i = 0; i < DECKSIZE; i++)
  {
    int value = -1;
    do
    {
      value = rand() % DECKSIZE;
    }
    while(drawn[value] != 0);

    /* mark value as used */
    drawn[value] = 1;

    /* debug statement */
    printf("%i\n", value);
```

```
    pdeck->card[i] = value;
  }
  pdeck->dealt = 0;
  return;
}
```

Save this file as `shuffle.c`. We have put a debug statement in this code so that when it runs, it will write out the card numbers it generates. This doesn't add to the functionality of our program, but it's crucial right now so that we can see what's happening. Since we're still just beginning our game, we have no other way to make sure our function is doing what we want it to. With the printf statement, we can see exactly what's happening right now so that when we move on to our next phase we know the deck is well shuffled. After we're satisfied it's working correctly, we can take that line out of our code. This technique to debug programs seems crude, but it does the trick with a minimum amount of fuss. We will discuss more sophisticated debuggers later.

Notice two things.

1. We pass in a parameter by it's address, which you can tell from the '&' (address of) operator. This passes the machine address of the variable on to the function so the function can change the variable itself. It's possible to program with global variables, but they should only be used only rarely. Pointers are an important part of C and you should understand them well.
2. We are using a function call from a new .c file. The operating system always looks for a function called 'main' and begins executing there. `shuffle.c` has no 'main' function and therefore can't be made into a stand alone executable. We must combine it with another program that does have a 'main' and calls the 'shuffle' function.

Run the command

```
gcc -c shuffle.c
```

and make sure it creates a new file named `shuffle.o`. Edit the game.c file, and on line 7, after the declaration of the deck_t variable `deck`, add the line

```
shuffle(&deck);
```

Now, if we try to create an executable the same way as before, we get an error

```
gcc -o game game.c
```

```
/tmp/ccmiHnJX.o: In function ‘main’:
/tmp/ccmiHnJX.o(.text+0xf): undefined reference to ‘shuffle’
collect2: ld returned 1 exit status
```

The compile succeeded because our syntax was correct. The link step failed because we didn't tell the compiler where the 'shuffle' function is located. What's the *link* and how do we tell the compiler where to find this function?

Link

The linker, `ld`, takes the object code created previously by `as` and turns it into an executable by the command

```
gcc -o game game.o shuffle.o
```

This will combine the two objects together and create the executable `game`.

The linker finds the `shuffle` function from the shuffle.o object and includes it in the executable. The real beauty of object files comes from the fact that if we want to use that function again, all we only have to include the "deck.h" file and link the `shuffle.o` object file into the new executable.

Code reuse like this happens all the time. The we didn't write the `printf` function we called above as a debug statement, the linker finds it's definition in the file we include with the `#include <stdlib.h>` and links to the object code stored in the C library (/lib/libc.so.6). This way we can use someone else's function that we know works correctly and worry about solving our own problems. This is why header files normally contain only the data and functions definitions, not function bodies. You normally create object files or libraries for the linker to put into the executable. A problem could occur with our code since we did not put any function definitions in our header. What can we do to make sure everything goes smoothly?

Two More Important Options

The `-Wall` option turns on all kinds of language syntax warnings to help us make sure that your code is correct and as portable as possible. When we use that option and compile our code we see something like:

```
game.c:9: warning: implicit declaration of function ‘shuffle’
```

This lets us know that we have a little more work to do. We need to put a line in a header file where we tell the compiler all about our `shuffle` function so it can do the checking it needs to do. It sounds like a hassle, but it separates the definition from the implementation and allows us to use our function anywhere just by including our new header and linking in our object code. We will put this one line in the deck.h file.

```
void shuffle(deck_t *pdeck);
```

That will get rid of that warning message.

Another common compiler option is optimization `-O#` (i.e. -O2). This tells the compiler what level of optimization you want. The compiler has a whole bag of tricks to make your code go just a little bit faster. For a tiny program like ours you won't notice any difference, but for larger programs it can speed things up quite a bit. You see it everywhere, so you should know what it means.

Debugging

As we all know, just because our code compiles doesn't mean that it works in the way we want it too. You can verify that all the numbers are used just one time by running

```
game | sort - n | less
```

and checking that nothing is missing. What do we do if there's a problem? How do we look under the hood and find the error?

You can check your code with a debugger. Most distributions provide the classic debugger, gdb. If the command line options overwhelm you like they do me, KDE offers a very nice front-end with KDbg. Other front ends exist, and they are very similar. To start debugging, you choose File->Executable and then find your `game` program. When you press the F5 or choose Execution->Run from the menu, you should see output in a separate window. What happens? We don't see anything in the window. Don't worry, KDbg isn't broken. The problem stems from the fact that we haven't put any debugging information into the executable, so KDbg can't tell us what's going on internally. The compiler flag `-g` puts the needed information into the object files. You must compile the object files (.o extension) with this flag, so the command now becomes

```
gcc -g -c shuffle.c game.c
gcc -g -o game game.o shuffle.o
```

This puts hooks into the executable that allow gdb and KDbg to figure out what's going on. Debugging is an important skill, it's well worth your time to learn how to use one well. The way debuggers help programmers is the ability to set a 'Breakpoint' in the source code. Try to set one now by right-clicking on the line with the call the the `shuffle` function. A little red circle should appear next to that line. Now, when you press F5 the program stops executing on that line. Press F8 to step *into* the shuffle function. Hey, now we're looking at the code from `shuffle.c`! We can control the execution step by step and see what's really going on. If you let the arrow hover over a local variable, you will see what it holds. Sweet. It's a lot better than those `printf` statements, isn't it?

Summary

This article presented a whirlwind tour of compiling and debugging C programs. We discussed the steps the compiler goes through and what options to pass gcc to make it do those steps. We touched on linking in shared libraries and ended with an introduction to debuggers. It takes a lot of work to really know what you're doing, but I hope this served to start you off on the right foot. You can find more information in the `man` and `info` pages for `gcc`, `as` and `ld`.

Writing code yourself teaches you the most. To practice, you could take the bare beginnings of the card game program in this article and write a blackjack game. Take the time to learn how to use a debugger. It's much easier to start with a GUI one like KDbg. If you add a little bit of functionality at a time, you'll be done before you know it. Remember, keep it running!

Here are some of the things you might need to create a full game.

- A definition of a card player (i.e. you could define the deck_t as a player_t).
- A function that deals a given number a cards to a given player. Remember to increment the

number of 'dealt' in the deck to keep track of where you need to deal the next card from. Remember to keep track of how many cards in the player's hand.
- Some user interaction to ask if the player wants another card.
- A function to print out a player's hand. The *card* is value % 13 (yielding 0 through 12), the *suit* is value / 13 (yielding 0 through 3).
- A function to determine the value of a player's hand. Ace's are card value zero and can be 1 or 11. Kings are value 12 and worth 10.

Links

- gcc GCC GNU Compiler Collection
- gdb GNU Debugger
- KDbg KDE's GUI Debugger
- Award Winning Compiler Hack Ken Thompson's great compiler hack

---

2005-01-14, generated by lfparser_pdf version 2.51