How to Create and Use a C Static Library

Lee Gaines Follow 4 min read · Oct 6, 2017

THIS POST NOW LIVES ON MY BLOG

http://lee.earth/posts/how-to-create-and-use-a-c-static-library/

“A library is not a luxury but one of the necessities of life” | img src: datalogics

In this post I will explain what a static library is, how they work, how to create one, and how to use a static library in a computer program.

Why use libraries?

Software is made up of functions. Functions are kind of like mini-programs that can be “called” by another program to perform some type of action such as a computation or modification of input.

Here’s a C program that uses a function called **sum()** to add two numbers together and stores the output in a variable called “result”:

```
int main(void)
{
    int x = 5;
    int y = 8;
    int result;

    result = sum(x, y);
    return (0);
}
```

In order for this program to compile and become usable, the compiler must know what and where the function **sum()** is.

A programmer could define and write the function **sum()** above the main function so the compiler finds it right away, within the program itself. It could look something like this:

```
int sum(int a, int b)
{
   return (a + b);
}
```

```
int main(void)
{
    int x = 5;
    int y = 8;
    int result;

    result = sum(x, y);
    return (0);
}
```

sum() is a very basic function, it just adds two variables and returns the result. As functions get more and more complex, they can become very long and make the main program hard to read. In addition, programs usually call *numerous* functions, so hard-coding functions above **main()** can get very messy.

This is where **static libraries** come in. You can think of a library as a collection of books. Each book is a function that can be read by anyone who has access to the library. Static libraries help keep programs clean and concise for humans but they do have some drawbacks, which we will touch on later.

How they work

Static libraries, also called "archives", are just collections of object files that contain functions. All the functions within the library are organized and indexed with a *symbol* and address, kind of like a table of contents, which makes it easier to find what you're looking for.

Static libraries are joined to the main module of a program during the linking stage of compilation before creating the executable file. After a successful link of a static library to the main module of a program, the executable file will contain *both* the main program and the library.

How to create them

Static libraries are created using some type of archiving software, such as **ar**. **ar** takes one or more object files (that end in .o), zips them up, and generates an archive file (ends in .a) — This is our "static library".

Before using ar, we must first have some object files to give to it. Perhaps we've written some functions in C that we want to include in our library. We can use the **-c** option with the GNU compiler (**gcc**) to stop the compiling process after the assembling stage, translating our files from .c to .o.

```
$ gcc -c sum.c   // produces a sum.o object file
```

Now that we have the object file(s), we can archive them and make a static library using **ar**.

```
$ ar -rc libforme.a sum.o
```

The command above will create a static library called "libforme.a". Inside of it will be our sum() function that has been translated to an object file via the gcc command earlier.

Note: the -rc options create the archive without a warning and replaces any pre-existing object files in the library with the same name.

Some archivers automatically organize and index the library, but in case indexing didn’t occur, we can use a command called ranlib to generate and store an index in the archive.

The index lists each symbol defined by a member of an archive that is a relocatable object file. To see a list of the symbols from object files we can use a command called **nm**.

```
$ nm libforme.a

// sample output

sum.o:

000000000000002e T sum
```

So now that we’ve created object files, zipped them in up an library and indexed it, we are ready to use our library.

How to use them

Let’s say we want to compile our C program from the beginning of this post; the one that uses **sum()** to store the value of two numbers:

```
int main(void)
{
    int x = 5;
    int y = 8;
    int result;

    result = sum(x, y);
    return (0);
}
```

When we try to compile the program we might see an error like this:

```
gcc my_program.c
// oops... //
/tmp/ccGLAk66.o: In function `main':
my_program.c:(.text+0x26): undefined reference to `sum'
collect2: error: ld returned 1 exit status
```

The compiler doesn't know what sum is. We need to tell it to look in our library:

```
gcc my_program.c -L. -lforme -o my_program
```

Let's break that down:

- **-L** says "look in directory for library files"
- **.** (the dot after 'L') represents the current working directory
- **-l** says "link with this library file"
- **forme** is the name of our library. Note that we omitted the "lib" prefix and ".a" extension. The linker attaches these parts back to the name of the library to create a name of a file to look for.
- **-o my_program** says "name the executable file my_program"

If everything worked out, the result will be an executable file called my_program that uses the sum() function that is contained in the libforme.a static library.

I hope you found this post helpful in understanding and appreciating static libraries. If you have any questions, comments or suggestions, feel free to hit me up on Twitter @eightlimbed.

Libraries Software Programming Compilers C Language

Written by Lee Gaines

82 followers · 22 following

Follow

Software Engineer

Responses (1)

Write a response

What are your thoughts?

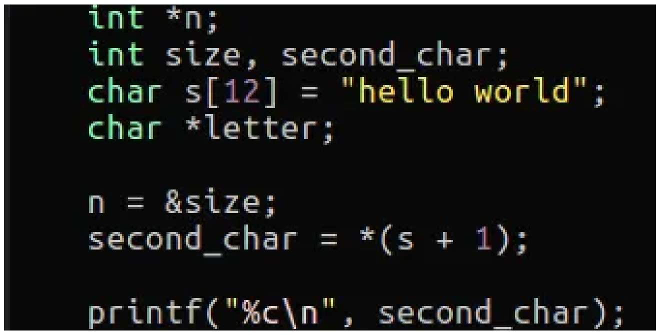
Gustavo Maldonado
Jan 18, 2022

Thanks for the clear and concise post. I followed the steps but ran into some trouble while compiling using flags (-Wall, -Wextra, -Werro), I get a warning: 'implicit declaration of function'.

How and where do I declare the function to avoid this warning?

Reply

More from Lee Gaines



```
int *n;
int size, second_char;
char s[12] = "hello world";
char *letter;

n = &size;
second_char = *(s + 1);

printf("%c\n", second_char);
```

Lee Gaines

Objects and Mutability in Python

THIS POST NOW LIVES ON MY BLOG

Jan 10, 2018 2 1

Lee Gaines

C Pointer Syntax in Plain English

THIS POST NOW LIVES ON MY BLOG:

Sep 28, 2017 96 1

Lee Gaines

What the ` export ` Command Does in Bash

THIS POST NOW LIVES ON MY BLOG

Jan 23, 2018 2

Lee Gaines

How and Why I Moved My Environment From Vagrant to...

Backstory

Mar 4, 2018 157 2

See all from Lee Gaines

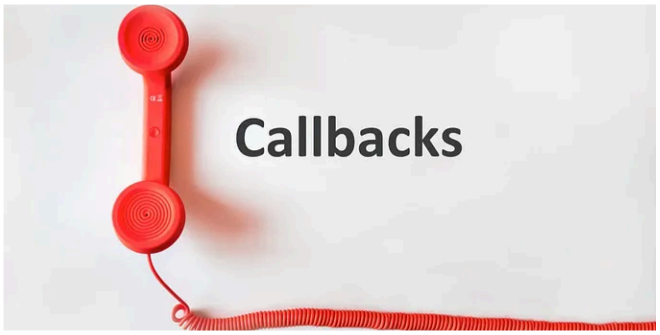
Recommended from Medium

Devlink Tips

Apple is quietly rewriting iOS and it's not in Swift or Objective-C

The hidden language shift happening inside Cupertino, why it matters, and what it means...

Sep 15 1.8K 47

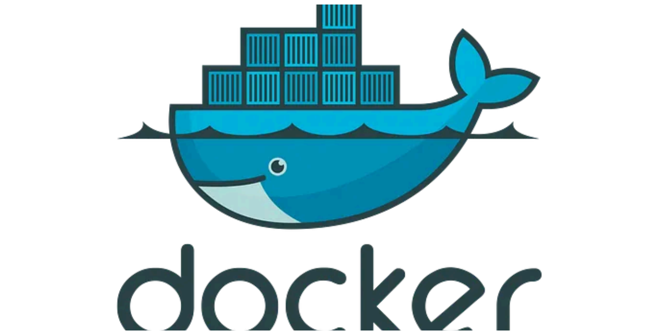


Oleh Slabak

Callback functions in C++

part#1

May 23 3 1



Abhinav

Docker Is Dead — And It's About Time

Docker changed the game when it launched in 2013, making containers accessible and...

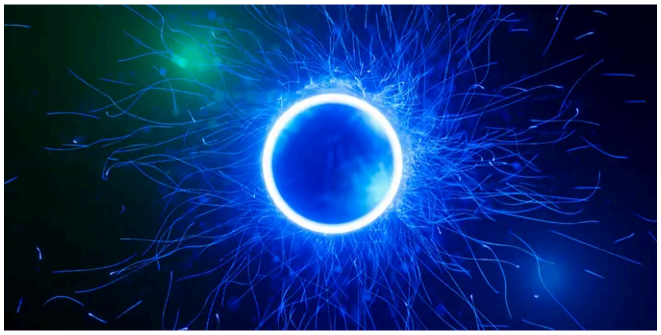
Jun 9 6.6K 182

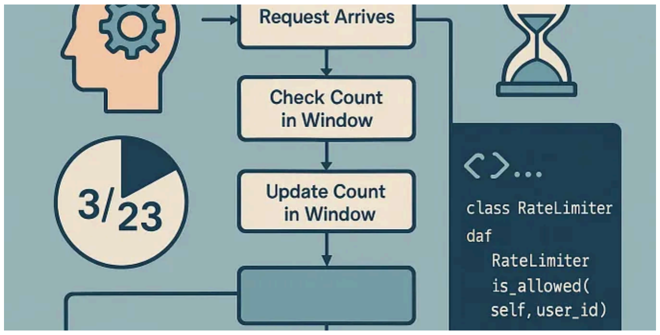
Dhanush kavin G

Why Learning C Programming is Essential for Understanding...

When we first step into the world of Operating Systems (OS), the amount of theory can feel...

Aug 26





Will Lockett

The AI Bubble Is About To Burst, But The Next Bubble Is Already...

Techbros are preparing their latest bandwagon.

Sep 14 11.3K 364

The Latency Gambler

I Interviewed 20+ Engineers. Here's Why Most Can't Code

Over the past year as a Senior Software Engineer at a B2B SaaS company, I've...

Sep 9 2.1K 68

See more recommendations