Introduction to C Programming Language

TechScribe

November 2022

LICENSING

This documentation is a work in progress, so some information may be incomplete or haven't been proofread.

Table of Contents

CHAPTER 1

GETTING STARTED

Before we get started on learning all about C Programming Language, we will need a few tools to assist our learning process. It should be mentioned that you only strictly really need a compiler and an editor to begin writing C language program, any other extra tools are there to help you organize your project, debug C program, and configure build process.

Over the course of this book, we will be covering a number of topics including the fundamentals of C language, basic of assembly programming (the book won't focus too much on this), SIMD programming, runtime compiler programming, and many more. If any topic is missing in this book, you can contact the author, TechScribe, at https://github.com/TechScribe-Deaf/Docs by creating an issue on the project repository.

To emphasize simplicity of managing C Project, we will be using Meson Build project which is an opinionated build system that strives to make building process simplier. While this tool may serves you well in some types of projects, it may not serves well at all in other projects so you may have to use CMake or other build tool to accomplish the tasks required. For instance, meson build won't allow in-source code generation in source code directories, it wanted to have all generated code stored in build directory. As I reiterates, Meson Build is a good tool to simplify the build process, it just not the most flexible build system in the world like CMake or build script is. It is a good tool to use for majority of this book where we will be covering the basic of C language.

We will be using LLDB and various frontend debugging software to assist the process in writing C Programming Language. Another component to debugging is to also simulates failures in our C program such as failure to allocate memory, so we will need to curate a number of utilities. We will be utilizing a number of techniques for unit testings and this is particularly important when we're writing in a programming language that lacks any safety guard rails that we might see in C++ or Rust. C language have it's benefits and drawbacks, we will iterate what each are and when you should consider writing C and when not to.

For majority of this book, we will not be covering Windows Operating System development environment. You generally may need to install Visual Studio Community Edition (not to be confused with Visual Studio Code which is an entirely different product) if licensing allows and to use MSVC build system if possible. If you still wish to use C language on Windows, you may need to install the following software: LLVM, Clang, MesonBuild, and your favorite editor. We won't be covering installation of such tools in this book as this book is oriented toward Linux distribution. I will be happy to udpate this documentation in the future to include Windows if time allows or if patreon goal is met, I am working on this documentation for free afterall.

REQUIRED TOOLS

1.1.1 Meson Build

Meson Build is a build system to simplify the process of compiling a number of source codes and linking process for variety of programming languages. It also simplify the process for mixing programming language in the same project as well.

It can be installed by typing the following command for your given Linux distribution (If it is not listed, you may need to install it from source: https://github.com/mesonbuild/meson):

Linux Distribution	Installation Command
Arch Linux	pacman -S meson
Debian/Ubuntu/PopOS	apt install meson
OpenSUSE/SUSE	zypper install meson
CentOS/Red Hat Enterprise Linux	yum install meson
Fedora	dnf install meson
NixOS	nix-env -iA nixos.meson

1.1.2 LLVM and Clang

LLVM is an open source compiler project that focuses on Compiler Intermediate Language to assembly code whereas Clang is also an open source compiler project that focuses on C/C++ source code to Compiler Intermediate Language. When used together, it allows compilation of C code to assembly code, but Clang have made that transparent anytime you compile C code. You can however emit Compiler Intermediate Language in Clang by adding "-S -emit-llvm" which would produces the following code for an example:

```
define dso_local i32 @main(i32 noundef %0, i8** noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i8**, align 8
  store i32 0, i32* %3, align 4
  store i32 %0, i32* %4, align 4
  store i8** %1, i8*** %5, align 8
  %6 = load i32, i32* %4, align 4
  %7 = zext i32 %6 to i64
  %8 = load i32, i32* %4, align 4
  %9 = icmp ne i32 %8, 2
  br i1 %9, label %10, label %12

10:                                               ; preds = %2
  %11 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([20 x i8], [20 x i8]* @.str, i64 0, i64 0))
  store i32 1, i32* %3, align 4
  br label %17

12:                                               ; preds = %2
  %13 = load i8**, i8*** %5, align 8
  %14 = getelementptr inbounds i8*, i8** %13, i64 1
  %15 = load i8*, i8** %14, align 8
  %16 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x i8], [4 x i8]* @.str.1, i64 0, i64 0), i8* noundef %15)
  store i32 0, i32* %3, align 4
  br label %17

17:                                               ; preds = %12, %10
  %18 = load i32, i32* %3, align 4
  ret i32 %18
}

declare i32 @printf(i8* noundef, ...) #1
```

Figure 1.1: LLVM IR Example

Linux Distribution	Installation Command
Arch Linux	pacman -S llvm clang
Debian/Ubuntu/PopOS	apt install llvm clang
OpenSUSE/SUSE	zypper install llvm clang
CentOS/Red Hat Enterprise Linux	yum install clang llvm
Fedora	dnf install clang llvm
NixOS	nix-env -iA nixos.llvm nix-env -iA nixos.clang

1.1.3 LLDB and GDB

LLDB is a debugger tool based on LLVM/Clang projects and GDB is also a debugger based on GCC projects, we will be using both tools throughout the book for debugging our software.

Linux Distribution	Installation Command
Arch Linux	pacman -S lldb
Debian/Ubuntu/PopOS	apt install lldb
OpenSUSE/SUSE	zypper install lldb
CentOS/Red Hat Enterprise Linux	yum install lldb
Fedora	dnf install lldb
NixOS	nix-env -iA nixos.lldb

1.1.4 Doxygen

Doxygen is a documentation generation tool that reading source code and generating documentation from that. It is useful for providing API reference documentation, number of call graphs, extended documentation on usage and tutorials, and number of other things. We will be using this throughout the chapter as we learn about C Language together.

Linux Distribution	Installation Command
Arch Linux	pacman -S doxygen
Debian/Ubuntu/PopOS	apt install doxygen
OpenSUSE/SUSE	zypper install doxygen
CentOS/Red Hat Enterprise Linux	yum install doxygen
Fedora	dnf install doxygen
NixOS	nix-env -iA nixos.doxygen

1.1.5 Git

Git is the center piece for managing different versions of our projects and it allows us to incorporate patches from other contributors. We will be using Git to version our works as we go through the project together. Git is not to be confused with Github, Github is a service that provide hosting for Git versioned projects to upload to, Git does not strictly requires Github nor does it require any other services. Git can even be used over email for an example, so it have always been a decentralized tool for programming.

Linux Distribution	Installation Command
Arch Linux	pacman -S git
Debian/Ubuntu/PopOS	apt install git
OpenSUSE/SUSE	zypper install git
CentOS/Red Hat Enterprise Linux	yum install git
Fedora	dnf install git
NixOS	nix-env -iA nixos.git

1.1.6 Valgrind

Valgrind is an analysis utility for detecting bugs in memory management and threading. It can also generates callgrind which reveals hot paths in code, this program can be extremely slow for most circumstances, but prove invaluable for detecting hard to find bugs.

Linux Distribution	Installation Command
Arch Linux	pacman -S valgrind
Debian/Ubuntu/PopOS	apt install valgrind
OpenSUSE/SUSE	zypper install valgrind
CentOS/Red Hat Enterprise Linux	yum install valgrind
Fedora	dnf install valgrind
NixOS	nix-env -iA nixos.valgrind

OPTIONAL TOOLS

1.2.1 Kate Editor

Kate is a software made under KDE foundation and it have supports similar to alternative editor called Visual Studio Code by Microsoft. Kate offers a direct support for Language Server Protocol without requiring significant extension to support autocompletion, type lookup, and other informations. Kate Editor is freely available for just about every platform and can be found on https://kate-editor.org/.

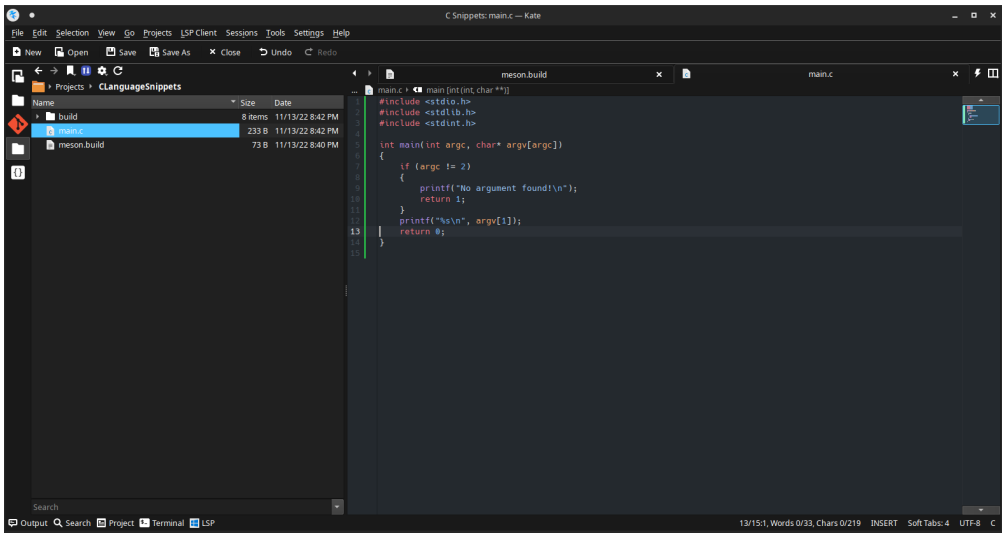

Figure 1.2: Kate Editor

It can be installed by typing the following command for your given Linux distribution (if it is not listed, you may either visit https://kate-editor.org/ for more informations or build from source.)

Linux Distribution	Installation Command
Arch Linux	pacman -S kate
Debian/Ubuntu/PopOS	apt install kate
OpenSUSE/SUSE	zypper install kate
CentOS/Red Hat Enterprise Linux	It may be recommended to use either flatpak or appimage distribution of Kate from https://kate-editor.org/get-it/
Fedora	dnf install kate
NixOS	nix-env -iA nixos.libsForQt5.kate

1.2.2 KDbg

KDbg is a frontend debugger for GDB and it is developed under KDE foundation. This tool is entirely optional and is used to ease the beginner in debugging codes.

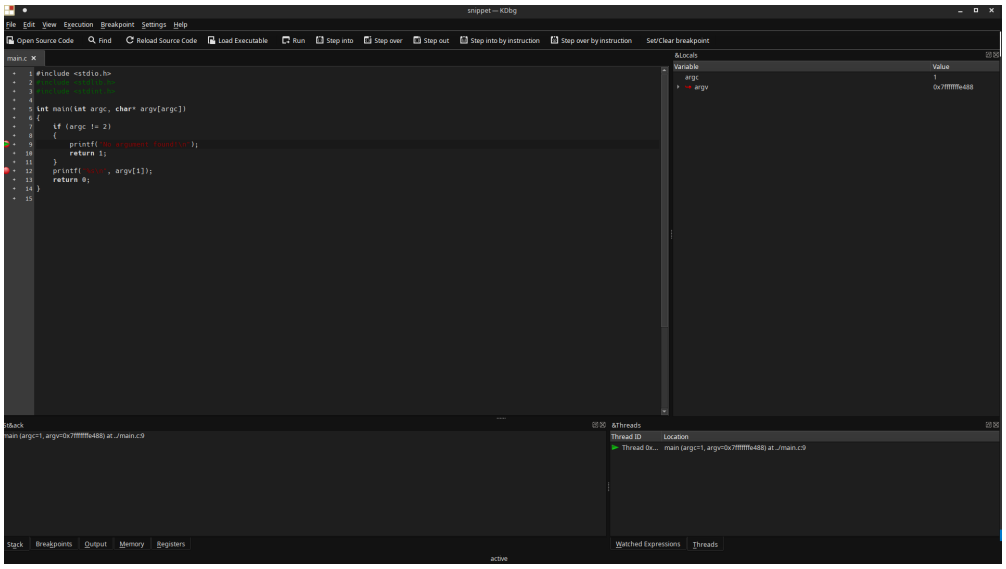

Figure 1.3: KDbg Debugger

Linux Distribution	Installation Command
Arch Linux	pacman -S kdbg
Debian/Ubuntu/PopOS	apt install kdbg
OpenSUSE/SUSE	zypper install kdbg
CentOS/Red Hat Enterprise Linux	yum install kdbg
Fedora	dnf install kdbg
NixOS	nix-env -iA nixos.kdbg

1.2.3 CppCheck

CppCheck is a static code analyzer utility that would scan your code for any errors/mistakes although it may results in false positives depending on code, this is still a useful utility none the less and can be proven invaluable for beginners.

Linux Distribution	Installation Command
Arch Linux	pacman -S cppcheck
Debian/Ubuntu/PopOS	apt install cppcheck
OpenSUSE/SUSE	zypper install cppcheck
CentOS/Red Hat Enterprise Linux	yum install cppcheck
Fedora	dnf install cppcheck
NixOS	nix-env -iA nixos.cppcheck

1.2.4 KCacheGrind

KCacheGrind is a callgrind visualizer utility to determine hot paths in your code. It can be a useful tool to identify slow code that could be optimized.

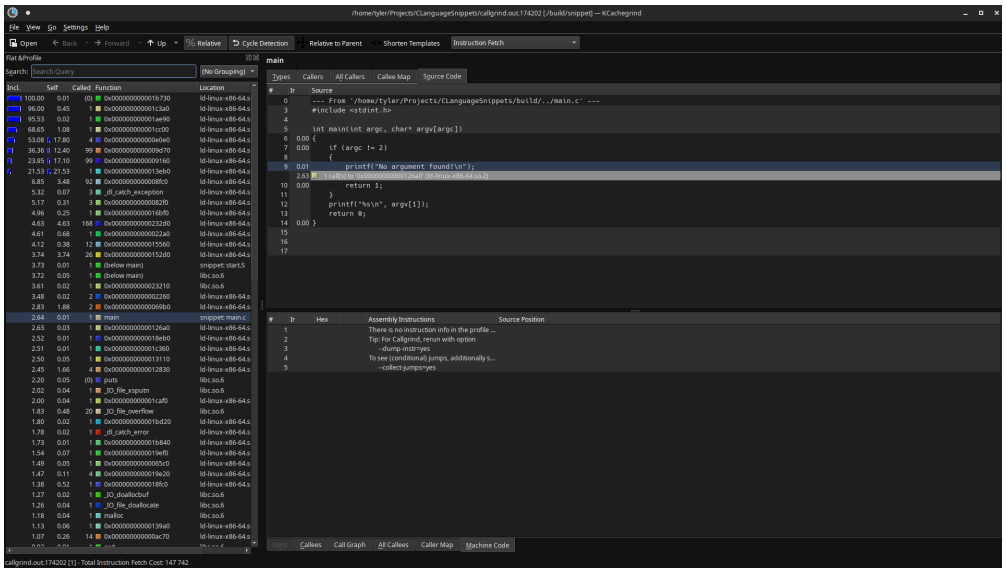

Figure 1.4: KCacheGrind

Linux Distribution	Installation Command
Arch Linux	pacman -S kcachegrind
Debian/Ubuntu/PopOS	apt install kcachegrind
OpenSUSE/SUSE	zypper install kcachegrind
CentOS/Red Hat Enterprise Linux	yum install kcachegrind
Fedora	dnf install kcachegrind
NixOS	nix-env -iA nixos.libsForQt5.kcachegrind

EXTERNAL RESOURCES

1.3.1 Godbolt Compiler Explorer

https://godbolt.org/ is an invaluable tool where it allows you to readily view the generated output from resulting code you submit across a range of compilers available on the website including pre-released compilers.

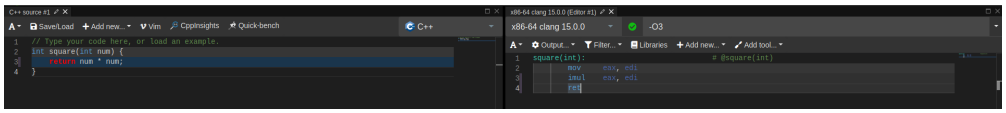

Figure 1.5: Godbolt Compiler Explorer

1.3.2 Tutorials Point

https://www.tutorialspoint.com/c_standard_library/index.htm serves as a useful reference for various functions that we'll be using in C language.

1.3.3 IBM z/OS C/C++ Language Reference

https://www.ibm.com/docs/en/zos/2.1.0?topic=cc-zos-xl-language-reference even though it's for a different operating system, z\OS, IBM provided a very useful reference for C11 standard that we will be using throughout this book especially on atomicity and multi-threaded programming.

1.3.4 Intel x86-64 Assembly Reference

https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html is a reference for Assembly Instructions, we will only lightly cover this topic when discussing SIMD, but this resource is available to you.

CHAPTER 2

THE BASIC

Setting Up The Environment

Over the course of this chapter, we will touch on the basic syntax for C Language as well as some of the compilation stages that we will see with our compilers. Let's create some folders for our projects, you can enter the following into the integrated terminal or your preferred terminal emulator program:

```
mkdir -p ~/Projects/chapter2
```

Then open either your favorite editor or Kate, and then go to File menu item and clicks on Open Folder and then navigate to our newly created folder, " /Projects/chapter2". Click on "New" or "New File" button and then write the followings:

```
project('chapter2', 'c')
src = [ 'main.c' ]
program = executable('chapter2', src)
```

And then save the newly created document as "meson.build" under " /Projects/chapter2" folder. Now create a new document by clicking on "New" or "New File" button once more and then enter the followings:

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

int main(int argc, char* argv[argc])
{
    printf("Hello world!\n");
    return 0;
}
```

Then save that newly created file as "main.c" under " /Projects/chapter2". Finally, in your integrated terminal or terminal emulator, you can run the following commands:

```
meson setup build
meson compile -C build
./build/chapter2
```

And you'll finally see an output appears:

```
[techscribe@SecureDev chapter2]$ ./build/chapter2
Hello world!
[techscribe@SecureDev chapter2]$
```

Figure 2.1: Chapter2 Executable Output

MESON BUILD

As you may have seen in the previous section, we have written a simple Meson build script that do three things:

1. Defines the project, naming it "chapter2" and that it is using C Language.
2. Defines an array of source code files to compile
3. Define an executable by naming it "chapter2" and to compile provided array of source code files

The project function is the first function that is going to be called in Meson before anything else to initialize meson. It allows you to provide the licensing, default options, required meson version, top level subdirectory path, and versioning as well as project name and language. Anytime your project get incorporated into another project, that project will read from this particular function when resolving dependency.

Meson Build system uses a declarative language that you can simply name something and assign it to holds a value and in this case, an array of source code files. This array variable is declared for the matter of convenience.

With array variable declared, you can pass in as an argument for executable function after naming it's executable and when executable is built, you can optionally assign it to a variable which can then be used for a more complex build steps such as generating codes.