

Introduction to C Programming Language

©2022 - TechScribe - All Rights Reserved.

TechScribe

November 2022

LICENSING

TechScribe is the author of this documentation and is the sole copyright owner of this documentation apart from fonts, external web resources listed or software used within this documentation.

The license of this documentation is granted to you under Creative Commons NonCommercial license 3.0 which you can find more information from <https://creativecommons.org/licenses/by-nc/3.0/legalcode> or by contacting TechScribe@linuxdev.app.

Each software and external resources are owned by it's respective copyright owners, Techscribe does not claim any ownership over those software and resources.

Font used in this documentation is Crimson Text (<https://fonts.google.com/specimen/Crimson+Text>) which is licensed under Open Font License which can be found at https://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&id=OFL.

Icons used for alerts and warnings is Noto Color Emoji (<https://fonts.google.com/noto/specimen/Noto+Color+Emoji>) which is licensed under Open Font License which can be found at https://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&id=OFL.

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

Table of Contents

Licensing	ii
1 Getting Started	1
1.1 Required Tools	2
1.1.1 Meson Build	2
1.1.2 LLVM and Clang	2
1.1.3 GDB and LLDB	3
1.1.4 Doxygen	3
1.1.5 Git	3
1.1.6 Valgrind	3
1.2 Optional Tools	4
1.2.1 Kate Editor	4
1.2.2 KDbg	5
1.2.3 Bear	5
1.2.4 CppCheck	6
1.2.5 KCacheGrind	6
1.3 External Resources	7
1.3.1 Godbolt Compiler Explorer	7
1.3.2 Tutorials Point	7
1.3.3 IBM z/OS C/C++ Language Reference	7
1.3.4 Intel x86-64 Assembly Reference	7
2 Tools for C Programming	9
2.1 Meson Build	10
2.2 Running CppCheck	10
2.2.1 With Bear Utility	10
2.2.2 Without Bear Utility	10
2.3 Using Debugger	12
2.3.1 Walking The Stacktrace	13
2.3.2 Printing Informations on Variable and Function Parameters	13
2.4 Frontend for Debugger	14
2.5 The Valgrind	15
2.5.1 Callgrind with Valgrind	16
2.6 KCachegrind	17
2.7 Documentation	18
2.7.1 Doxygen	18
2.8 Version Control	19

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 simpler. 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 update 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	<code>pacman -S meson</code>
Debian/Ubuntu/PopOS	<code>apt install meson</code>
OpenSUSE/SUSE	<code>zypper install meson</code>
CentOS/Red Hat Enterprise Linux	<code>yum install meson</code>
Fedora	<code>dnf install meson</code>
NixOS	<code>nix-env -iA nixos.meson</code>

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

Figure 1.1: LLVM IR Example

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

1.1.3 GDB and LLDB

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	<code>pacman -S lldb gdb</code>
Debian/Ubuntu/PopOS	<code>apt install lldb gdb</code>
OpenSUSE/SUSE	<code>zypper install lldb gdb</code>
CentOS/Red Hat Enterprise Linux	<code>yum install lldb gdb</code>
Fedora	<code>dnf install lldb gdb</code>
NixOS	<code>nix-env -iA nixos.lldb</code> <code>nix-env -iA nixos.lldb</code>

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	<code>pacman -S doxygen</code>
Debian/Ubuntu/PopOS	<code>apt install doxygen</code>
OpenSUSE/SUSE	<code>zypper install doxygen</code>
CentOS/Red Hat Enterprise Linux	<code>yum install doxygen</code>
Fedora	<code>dnf install doxygen</code>
NixOS	<code>nix-env -iA nixos.doxygen</code>

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	<code>pacman -S git</code>
Debian/Ubuntu/PopOS	<code>apt install git</code>
OpenSUSE/SUSE	<code>zypper install git</code>
CentOS/Red Hat Enterprise Linux	<code>yum install git</code>
Fedora	<code>dnf install git</code>
NixOS	<code>nix-env -iA nixos.git</code>

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	<code>pacman -S valgrind</code>
Debian/Ubuntu/PopOS	<code>apt install valgrind</code>
OpenSUSE/SUSE	<code>zypper install valgrind</code>
CentOS/Red Hat Enterprise Linux	<code>yum install valgrind</code>
Fedora	<code>dnf install valgrind</code>
NixOS	<code>nix-env -iA nixos.valgrind</code>

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/>.

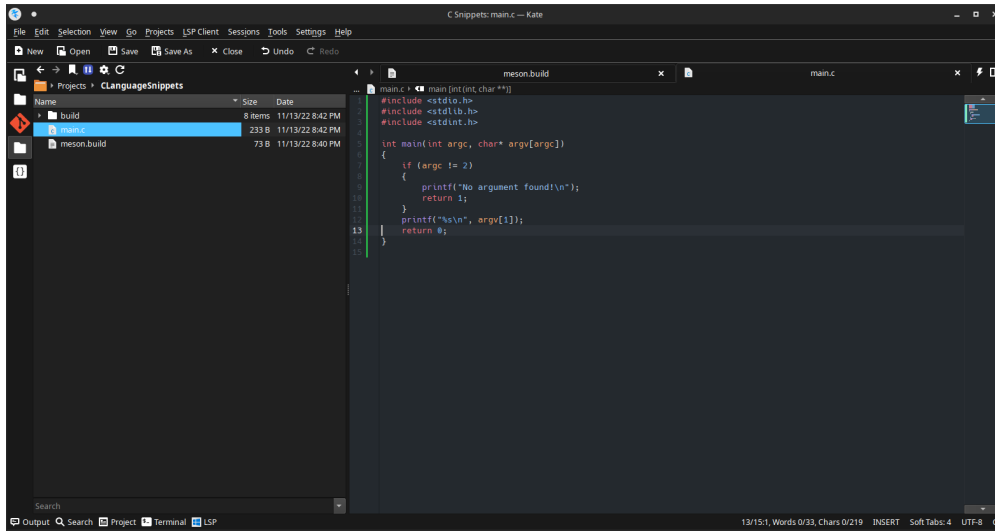


Figure 1.2: Kate Editor

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	<code>pacman -S kate</code>
Debian/Ubuntu/PopOS	<code>apt install kate</code>
OpenSUSE/SUSE	<code>zypper install kate</code>
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	<code>dnf install kate</code>
NixOS	<code>nix-env -iA nixos.libsForQt5.kate</code>

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.

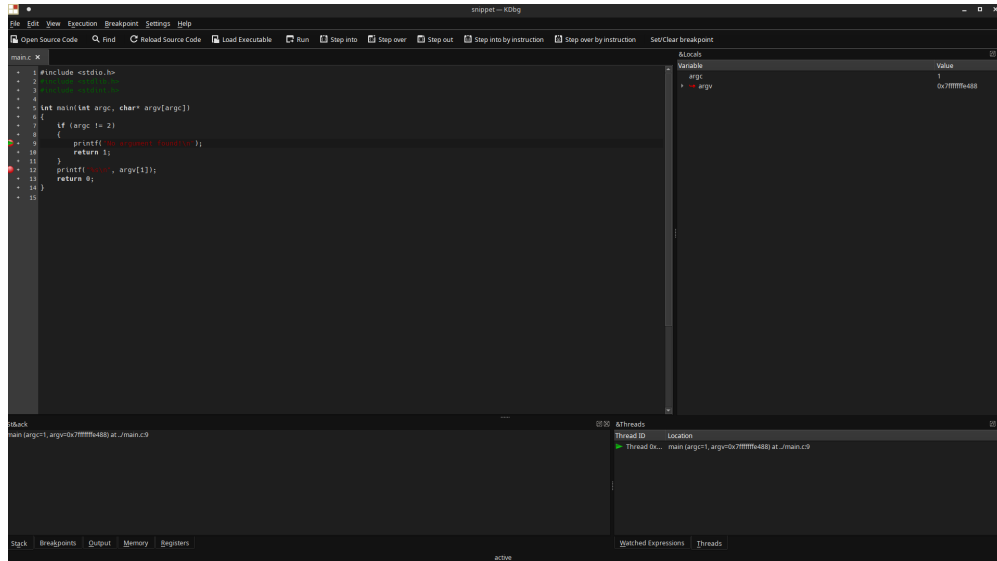


Figure 1.3: KDbg Debugger

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

1.2.3 Bear

Bear is a utility that overrides CC and CXX environment variable to record all of the compiler arguments and compile those commands into a command compilation database as a JSON file. We use bear utility to allow us to run various utility such as CppCheck tool. Meson provides something similar to this, but the command database is stored under specified build directory, so there's a number of inconvenience that can result from this. Bear utility is optional in this case.

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

1.2.4 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

1.2.5 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.

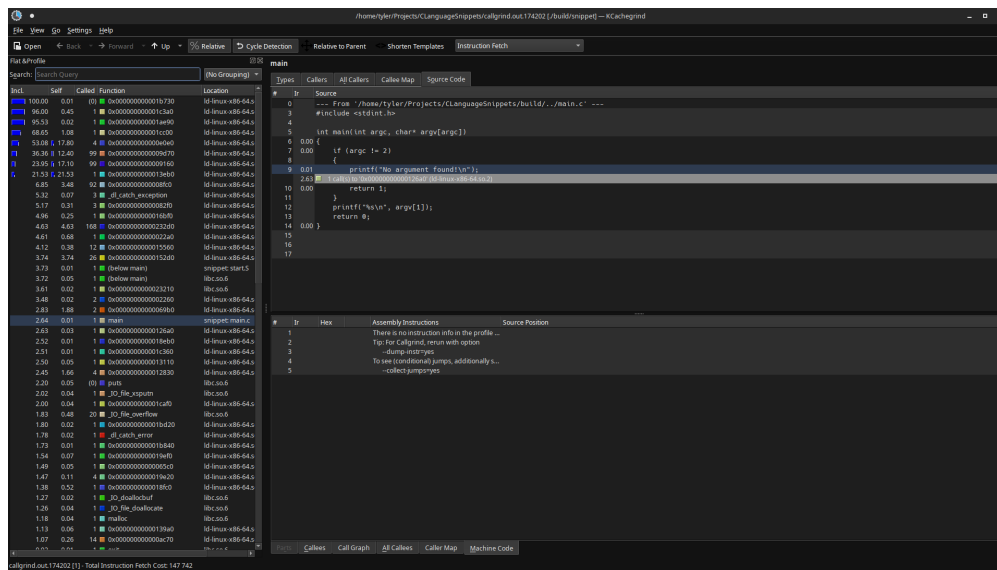


Figure 1.4: KCacheGrind

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.

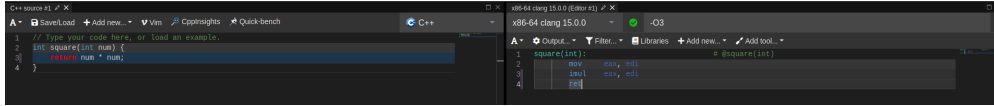


Figure 1.5: Godbolt Compiler Explorer

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.

TOOLS FOR C PROGRAMMING

Setting Up The Environment

Over the course of this chapter, we will first set up our environment for debugging, static code analysis, build script, and few others to help provide you the tools you need to learn C Programming Language. First, let's set up our very basic C project:

```
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:

```
1 project('chapter2', 'c')
2 src = [ 'main.c' ]
3 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:

```
1 #include <stdint.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 int main(int argc, char* argv[argc])
6 {
7     printf("Hello world!\n");
8     return 0;
9 }
```

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:

```
CC=clang 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.

When setting up Meson project in the terminal, we need to specify which compiler that we'll be using for C Language compilation. "CC" Environment variable specify the C language compiler and so by providing that variable to Meson setup, it will be configured to use clang going forward when you run:

```
CC=clang meson setup build
```

When compiling a meson based project after setting up the build directory, Meson would simply invoke ninja build process when running the following command:

```
meson compile -C build
```

The "-C" build parameter simply specifies which build directory it should build in and it get passed to Ninja build program.

After building, it would have binary available under build directory named, "chapter2".

RUNNING CPPCHECK

If you have CppCheck installed, you could run the CppCheck in two separate ways:

2.2.1 With Bear Utility

```
meson compile -C build --clean
bear -- meson compile -C build
cppcheck --enable=all --std=c11 --suppress=missingInclude \
    --project=compile_commands.json
```

2.2.2 Without Bear Utility

```
meson compile -C build --clean
meson compile -C build
cd ./build
cppcheck --enable=all --std=c11 --suppress=missingInclude \
    --project=compile_commands.json
```

To see our cppcheck in action, let's change the code of our "main.c" file content to as followed:

```

1 #include "stdint.h"
2 #include "stdlib.h"
3 #include "stdio.h"
4
5 int error() {
6     void* ptr = malloc(1024); // Intentionally cause a memory leak!
7     return 0;
8 }
9
10 int main(int argc, char* argv[])
11 {
12     error();
13     printf("Hello world! %s\n", argv[9]);
14     return 0;
15 }

```

And when repeating either commands above for running cppcheck, we'll have the following errors generated from cppcheck:

```

[techscribe@SecureDev build]$ cppcheck --enable=all --std=c11 --suppress=missingInclude --project=compile_commands.json
Checking /home/techscribe/Projects/chapter2/main.c ...
Checking /home/techscribe/Projects/chapter2/main.c: _FILE_OFFSET_BITS=64...
/home/techscribe/Projects/chapter2/main.c:7:5: error: Memory leak: ptr [memleak]
    return 0;
    ^
/home/techscribe/Projects/chapter2/main.c:6:15: style: Variable 'ptr' is assigned a value that is never used. [unreadVariable]
    void* ptr = malloc(1024); // Intentionally cause a memory leak!
    ^
/home/techscribe/Projects/chapter2/main.c:6:17: style: Variable 'ptr' is allocated memory that is never used. [unusedAllocatedMemory]
    void* ptr = malloc(1024); // Intentionally cause a memory leak!
    ^

```

Figure 2.2: CppCheck Errors Result

This utility can be a handy tool to assist you in finding errors/mistakes in your code as you learn C Language Programming.



CppCheck will generate warnings about the missing header includes especially those that are in standard library headers. You can safely ignore that or provide "--suppress=missingInclude" to suppress this warning.

USING DEBUGGER

At some points during the course of your learning and professional programming, you'll have to use debugger to figure out the errors in your code. There are two main debuggers that we often use on Linux, GDB and LLDB. So for the demonstration of this section, we'll first update our code in "main.c" file content oncemore to as followed:

```
1 #include "stdint.h"
2 #include "stdlib.h"
3 #include "stdio.h"
4
5 int error() {
6     int* badPtr = NULL;
7     return *badPtr;
8 }
9
10 int main(int argc, char* argv[])
11 {
12     error();
13     printf("Hello world! %s\n", argv[9]);
14     return 0;
15 }
```

Now that we've updated our "main.c" file, we can go ahead and recompile the project. Make sure you have set the current directory to "chapter2" directory first:

```
cd ~/Projects/chapter2
meson compile -C build
./build/chapter2
```

Once you run "chapter2" executable, you'll notice an error appears, "Segmentation fault (core dumped)", but it doesn't detail which code or line the error actually occurs at, so let's find out by running GDB debugger.

```
gdb -w ./build/chapter2
```

You'll see a few prompts, the first prompt will be the legal greeting, you can type in "c" and then press enter to continue without paging or to press enter to read through documentation as needed. Now you're running GDB, you can enter a command, "run" to run chapter2 executable and GDB will work to intercept any error that occurs in your program. The next prompt will asks if you want to download debuginfo for this session, you can choose yes and then you'll see the followings:

```
https://debuginfod.archlinux.org
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading 1.07 MB separate debug info for /lib64/ld-linux-x86-64.so.2
Downloading 9.93 MB separate debug info for /usr/lib/libc.so.6
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0x000055555555149 in error () at ../main.c:7
7       return *badPtr;
(gdb) □
```

Figure 2.3: GDB Debugger Result

2.3.1 Walking The Stacktrace

Sometime we want to see further up the chain like what code is calling into this function for an example, so we can run a command called a "Backtrace" or simply "bt". Enter the following and then press enter:

```
bt
```

And then we have the following information:

```
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading 1.07 MB separate debug info for /lib64/ld-linux-x86-64.so.2
Downloading 9.93 MB separate debug info for /usr/lib/libc.so.6
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0x000055555555149 in error () at ../main.c:7
7       return *badPtr;
(gdb) bt
#0  0x000055555555149 in error () at ../main.c:7
#1  0x000055555555166 in main (argc=1, argv=0x7fffffffef298) at ../main.c:12
(gdb) █
```

Figure 2.4: GDB Debugger Backtrace

This is what is known as the Stacktrace and we can use this information to potentially identify the problem that lead to this error.

2.3.2 Printing Informations on Variable and Function Parameters

As mentioned before, we can walk up the stacktrace as we want, when you print a list of "Frames", you can see an index following after the # on the left hand side of each frame. We can select the frame within our main function by entering the following command:

```
select-frame 1
```

And then we can obtain information about our parameters for an example by entering the following command after selecting frame:

```
info args
```

And then we'll have the following output from GDB:

```
(gdb) info args
argc = 1
argv = 0x7fffffffef298
(gdb) █
```

Figure 2.5: GDB Debugger Information on Function Arguments

FRONTEND FOR DEBUGGER

One of the frontend we'll use for GDB debugger is KDbg, it is relatively more straightforward especially those who are new to programming in C. You can simply run KDbg software under your application launcher and under Development category. Once KDbg is opened, you can click on "Load Executable" button and then search for the compiled binary at " /Projects/chapter2/build/chapter2" and after loading that executable, you'll see source code for that program comes up as "main.c" tab. You can then click on "Run" button to execute the program and you'll immediately see program stops at the offending line 7.

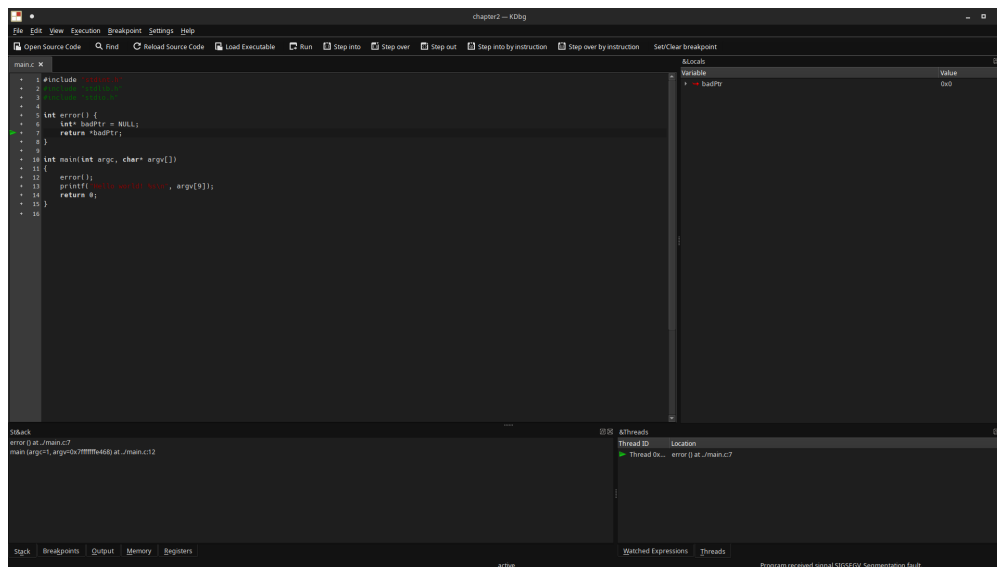


Figure 2.6: KDbg

On the right side view, you'll see a number of variables listed under "&Local" and bottom left window is the Stacktrace that you can select to see different sets of variables, registers, and other informations. And the right bottom view is the threads monitoring, you can switch stacktrace to viewing other thread apart from the one that offending code is on.

THE VALGRIND

Valgrind is a multitude of different analysis tools and it's generally known for memory analysis. It also can be used to generate what is known as the Callgrind, which can be used to determine the hot paths in code. Enter the following code in "main.c" file content:

```
1 #include "stdint.h"
2 #include "stdlib.h"
3 #include "stdio.h"
4
5 int* demo(int value)
6 {
7     int* ptr = (int*)malloc(sizeof(int));
8     *ptr = value;
9     return ptr;
10 }
11 int main(int argc, char* argv[])
12 {
13     int* ptr = demo(123);
14     free(ptr);
15     return 0;
16 }
```

And then finally run the following commands by first rebuilding the chapter2 binary and then run valgrind memory analysis on the binary:

```
cd ~/Projects/chapter2
meson compile -C build
valgrind ./build/chapter2
```

We'll have the following output:

```
==2704124== HEAP SUMMARY:
==2704124==    in use at exit: 0 bytes in 0 blocks
==2704124==   total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==2704124==
==2704124== All heap blocks were freed -- no leaks are possible
==2704124==
==2704124== For lists of detected and suppressed errors, rerun with: -s
==2704124== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
[techscribe@SecureDev chapter2]$
```

Figure 2.7: Valgrind Success Output

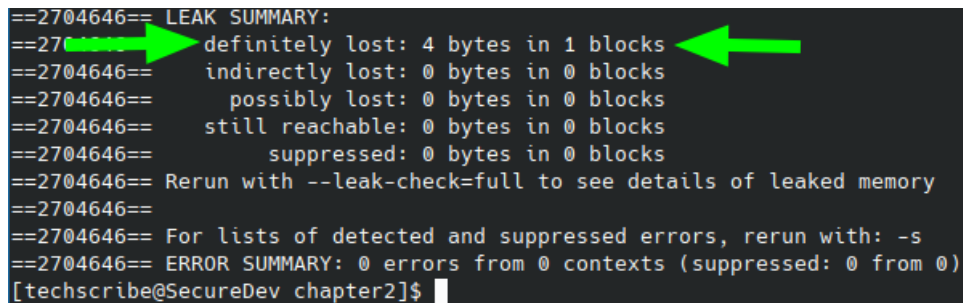
As expected from our small snippet, it will allocate a small block of memory and then responsibly free that memory by the end of the program. Now let's suppose we didn't do that instead and comment out the "free(ptr)" by adding double slashes before "free(ptr)". Update "main.c" file content to reflect as followed:

```
1 #include "stdint.h"
2 #include "stdlib.h"
3 #include "stdio.h"
4
5 int* demo(int value)
6 {
7     int* ptr = (int*)malloc(sizeof(int));
8     *ptr = value;
9     return ptr;
10 }
11 int main(int argc, char* argv[])
12 {
13     int* ptr = demo(123);
14     //free(ptr);
15     return 0;
16 }
```

Then run the following commands into terminal:

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

And we'll have the following output generated from valgrind:



```
==2704646== LEAK SUMMARY:
==2704646== definitely lost: 4 bytes in 1 blocks
==2704646== indirectly lost: 0 bytes in 0 blocks
==2704646== possibly lost: 0 bytes in 0 blocks
==2704646== still reachable: 0 bytes in 0 blocks
==2704646== suppressed: 0 bytes in 0 blocks
==2704646== Rerun with --leak-check=full to see details of leaked memory
==2704646== For lists of detected and suppressed errors, rerun with: -s
==2704646== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
[techscribe@SecureDev chapter2]$
```

Figure 2.8: Valgrind Memory Leak Output

As the green arrows indicates, valgrind is able to detect that we have a definite memory leak of about 4 bytes which is the same size allocated in our program. This is one of the many uses of Valgrind, it helps identify memory leak whether it occurs in your program or not.

2.5.1 Callgrind with Valgrind

Another tool available to you from valgrind is call a Callgrind, as explained before, it allows us to detect hot paths in our code. First, let's generate a callgraph output file that we can then analyze later. Run the following commands in terminal:

```
valgrind --tool=callgrind --dump-instr=yes --collect-jumps=yes ./build/chapter2
```

Now you'll find a newly created file in " /Projects/chapter2/callgrind.out.number" where number extension is generated from valgrind. Next, we'll use another set of tools to analyze this callgraph file called the KCachegrind.

KCACHEGRIND

As explained before, KCachegrind will be the tool we use to analyze Callgraph file, first by opening KCachegrind which should be listed in your application launcher under Development category. Once the program is open, click on "Open" button and navigate to " /Projects/chapter2/callgrind.out.number" and then it will loads something similar to this screenshot:

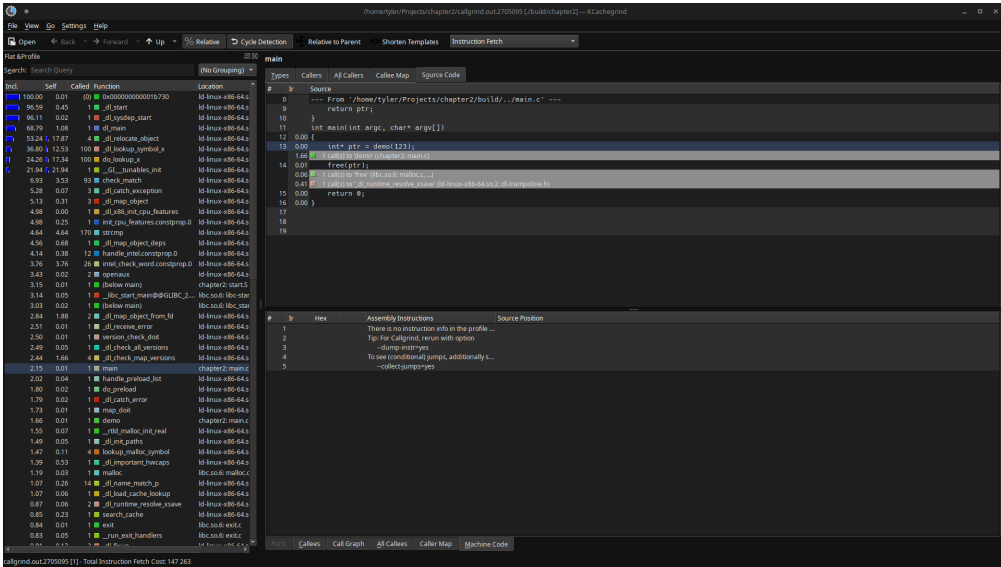


Figure 2.9: KCachegrind

KCachegrind offers a comprehensive suite of analysis and views that you can use, the bottom tabs listed, "Parts", "Callees", "Call Graph", "All Callees", "Caller Map", "Machine Code" Under "Call Graph", you can find a graph of which function calls which which can offers you an useful insight in how your program works.

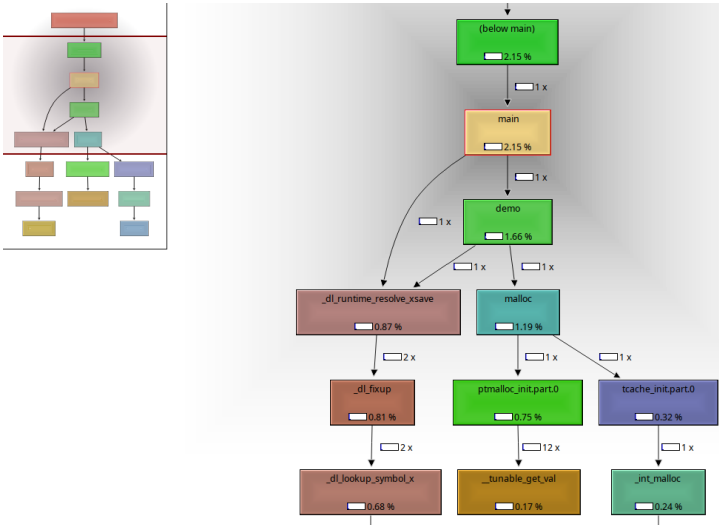


Figure 2.10: The Callgraph

While we only touch on this very lightly throughout the course of this book, you can learn more about KCachegrind by navigating to "Help" menu item and click on "KCachegrind Handbook" which would offers you comprehensive amount of information you need to effectively use KCachegrind.

DOCUMENTATION

2.7.1 Doxygen

One of the most common documentation is Doxygen, it allows us to document our code within source code and outside of it as well. It is commonly used to generate API Reference Documentation accompanying with the ability to add custom pages and generating callgraphs as well as incorporating diagrams as needed.

First, let's update our "main.c" file content to as followed:

```

1 #include "stdint.h"
2 #include "stdlib.h"
3 #include "stdio.h"
4
5 /// @brief A structure that represents the information
6 ///      about a human being, its name and age.
7 typedef struct Human {
8     /// @brief Name of a human, should not exceed 512 characters.
9     char name[512];
10    /// @brief Age of a human represented in number of years
11    ///      after birth, age value cannot be below 0.
12    uint age;
13 } Human;
14
15 int main(int argc, char* argv[])
16 {
17     Human TechScribe = (Human){
18         .age = 50,
19         .name = "TechScribe"
20     };
21
22     printf("This human being name is %s and it is %u years old.\n",
23         TechScribe.name, TechScribe.age);
24     return 0;
25 }
```

As you may have noticed the new comment lines following triple backslashes and "@brief", it is used to let Doxygen know to read for this source code and that those comments should be parsed to have documentation generated from those comments. The "@brief" is simply a way to let doxygen know that it is a shorthand summary of a given type, variable or function.

Now that we're ready to use Doxygen, we need to first generate a template doxygen configuration file by running the following command in terminal:

```
doxygen -g chapter2.doxyfile
```

You'll notice a newly generated "chapter2.doxyfile", you can optionally edit it or leave it as is for this demonstration. You can proceed to call Doxygen on the generated doxyfile configuration by running the following command in terminal:

```
doxygen chapter2.doxyfile
```

You'll see two directories appear, html and latex directories. Navigate to html and open with your favorite web browser, and you'll see a relatively blank page with Main Page and Classes tab. Navigate to classes tab and click on "Class list" and you'll see "Human" structure listed with its fields annotated.

VERSION CONTROL

Version Control is essentially a tool that allows you to manage the many versions and revisions of your code as well as collaborating with other people who may contribute to your code. Git is one of the most popular version control software in the programming industry, so this book will be covering it's many uses. If this is your first time using Git, you may want to configure your email and username to be used in git by running the following commands in terminal with both quoted email and name changed to your own:

```
git config --global user.email "YourEmail@example.com"
git config --global user.name "Your name"
```

You can initialize the repository by running the following command:

```
git init
```

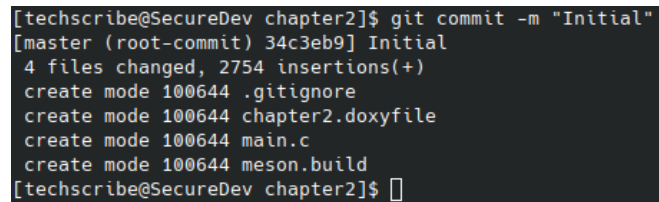
One of the first thing you may want to make before committing your files to git repository is to create a git ignore file which is used to list all of the files or directories you wish to be excluded from git history. Create a new ".gitignore" file, and make sure that the period in front of gitignore is included in it's filename under " /Projects/chapter2" folder and then enter the followings as ".gitignore" file content and then save:

```
build/
.cache/
```

Now you're ready to add files to git history by running the following commands:

```
git add .
git commit -m "Initial"
```

You will see the following output:



```
[techscribe@SecureDev chapter2]$ git commit -m "Initial"
[master (root-commit) 34c3eb9] Initial
4 files changed, 2754 insertions(+)
create mode 100644 .gitignore
create mode 100644 chapter2.doxyfile
create mode 100644 main.c
create mode 100644 meson.build
[techscribe@SecureDev chapter2]$
```

Figure 2.11: Git Example

This book will only lightly cover the basic use of Git version control, so you may need to read the manual and try using an interactive tutorial at <http://pcottle.github.io/learnGitBranching/>.

