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

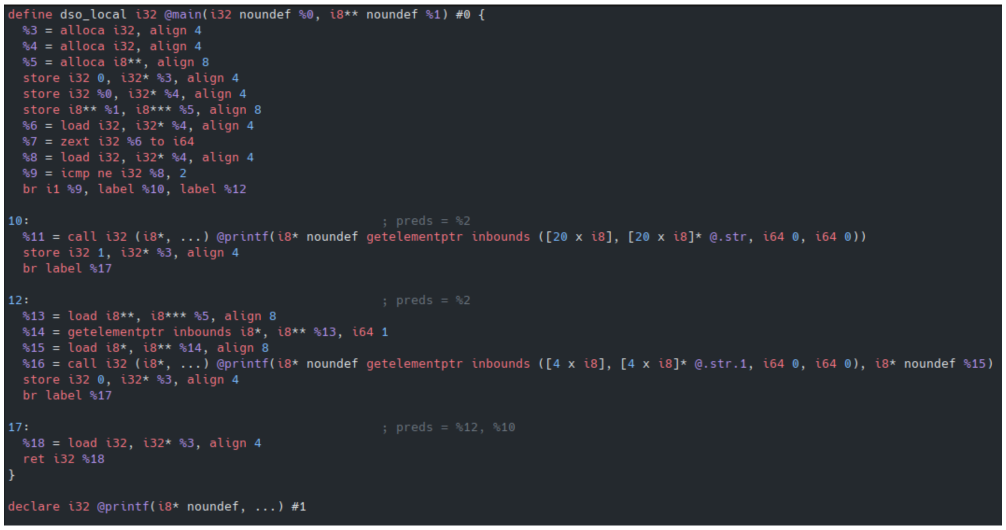
Meson Build is a build system to simplify the process of compiling a number of source codes and linking process for variety of programming languages. It also simplify the process for mixing programming language in the same project as well.

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

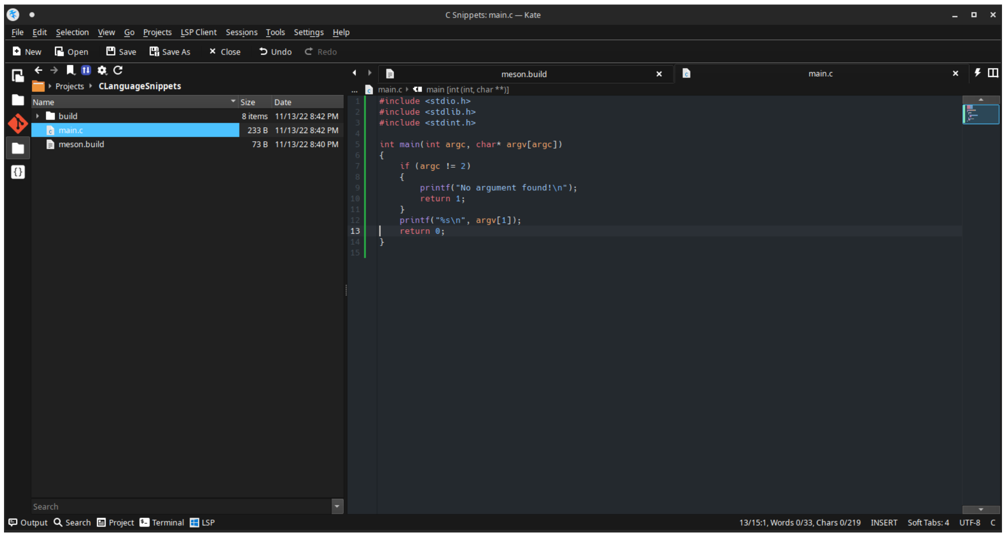
Valgrind is an analysis utility for detecting bugs in memory management and threading. It can also generates callgrind which reveals hot paths in code, this program can be extremely slow for most circumstances, but prove invaluable for detecting hard to find bugs.

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/.


C Snippets: main.c — Kate
File Edit Selection View Go Projects LSP Client Sessions Tools Settings Help
New Open Save Save As Close Undo Redo
Projects CLanguageSnippets
meson.build
main.c
main.c main [int (int, char **)]
Search
Output Search Project Terminal LSP
13/15:1, Words 0/33, Chars 0/219 INSERT Soft Tabs: 4 UTF-8 C

Name	Size	Date
build	8 items	11/13/22 8:42 PM
main.c	233 B	11/13/22 8:42 PM
meson.build	73 B	11/13/22 8:40 PM

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

int main(int argc, char* argv[argc])
{
    if (argc != 2)
    {
        printf("No argument found!\n");
        return 1;
    }
    printf("%s\n", argv[1]);
    return 0;
}
```

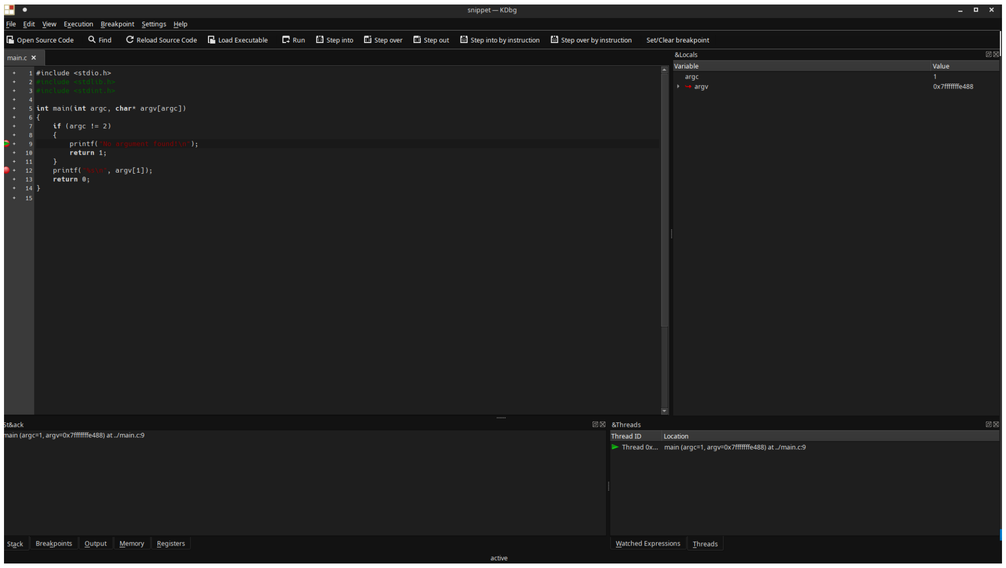
Figure 1.2: Kate Editor

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.



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

int main(int argc, char* argv[argc])
{
    if (argc != 2)
    {
        printf("No argument found!\n");
        return 1;
    }
    printf("%s\n", argv[1]);
    return 0;
}
```

&Locals

Variable	Value
argc	1
argv	0x7fffffffe488

St&ack

```
main (argc=1, argv=0x7fffffffe488) at ../main.c:9
```

&Threads

Thread ID	Location
Thread 0x...	main (argc=1, argv=0x7fffffffe488) at ../main.c:9

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 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 inconvience that can results from this. Bear utility is optional in this case. Kate editor with clangd for LSP support require "compile_commands.json" file to be based on the project toplevel rather than in the build directory, so it may be recommended to use bear utility for running "bear – meson compile -C build" at least once to generates usable "compile_commands.json" file and that it should be regenerated everytime a new source file or header file is created.

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

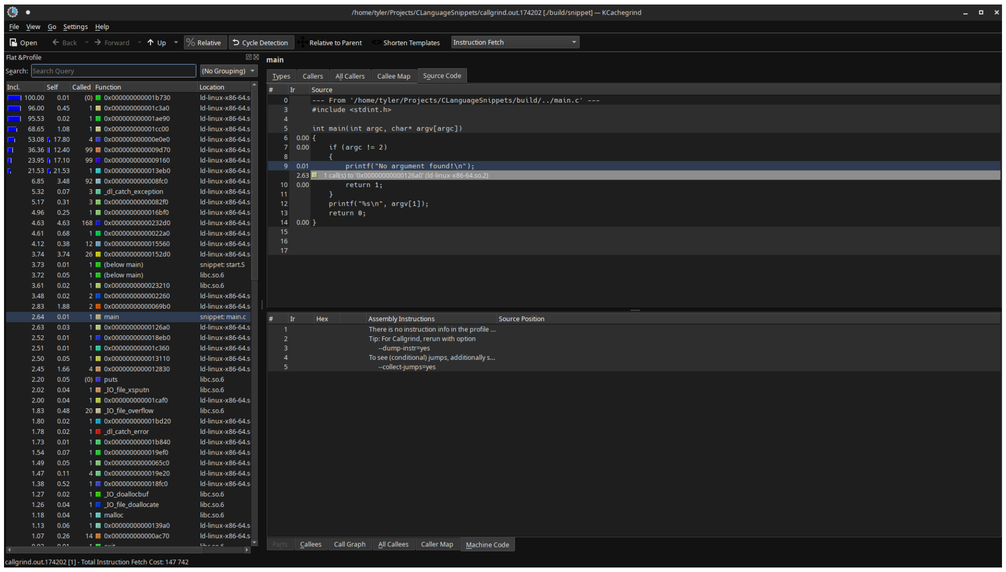
1.2.4 CppCheck

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.



/home/tyler/Projects/CLanguageSnippets/callgrind.out.174202 [./build/snippet] — KCachegrind

Incl.	Self	Called	Function	Location
100.00	0.01	(0)	0x000000000001b730	ld-linux-x86-64.s
96.00	0.45	1	0x000000000001c3a0	ld-linux-x86-64.s
95.53	0.02	1	0x000000000001ae90	ld-linux-x86-64.s
68.65	1.08	1	0x000000000001cc00	ld-linux-x86-64.s
53.08	17.80	4	0x000000000000e0e0	ld-linux-x86-64.s
36.36	12.40	99	0x0000000000009d70	ld-linux-x86-64.s
23.95	17.10	99	0x0000000000009160	ld-linux-x86-64.s
21.53	21.53	1	0x0000000000013eb0	ld-linux-x86-64.s
6.85	3.48	92	0x0000000000008fc0	ld-linux-x86-64.s
5.32	0.07	3	_dl_catch_exception	ld-linux-x86-64.s
5.17	0.31	3	0x00000000000082f0	ld-linux-x86-64.s
4.96	0.25	1	0x0000000000016bf0	ld-linux-x86-64.s
4.63	4.63	168	0x00000000000232d0	ld-linux-x86-64.s
4.61	0.68	1	0x00000000000022a0	ld-linux-x86-64.s
4.12	0.38	12	0x0000000000015560	ld-linux-x86-64.s
3.74	3.74	26	0x00000000000152d0	ld-linux-x86-64.s
3.73	0.01	1	(below main)	snippet: start.S
3.72	0.05	1	(below main)	libc.so.6
3.61	0.02	1	0x0000000000023210	libc.so.6
3.48	0.02	2	0x0000000000002260	ld-linux-x86-64.s
2.83	1.88	2	0x00000000000069b0	ld-linux-x86-64.s
2.64	0.01	1	main	snippet: main.c
2.63	0.03	1	0x00000000000126a0	ld-linux-x86-64.s
2.52	0.01	1	0x0000000000018eb0	ld-linux-x86-64.s
2.51	0.01	1	0x000000000001c360	ld-linux-x86-64.s
2.50	0.05	1	0x0000000000013110	ld-linux-x86-64.s
2.45	1.66	4	0x0000000000012830	ld-linux-x86-64.s
2.20	0.05	(0)	puts	libc.so.6
2.02	0.04	1	_IO_file_xsputn	libc.so.6
2.00	0.04	1	0x000000000001caf0	ld-linux-x86-64.s
1.83	0.48	20	_IO_file_overflow	libc.so.6
1.80	0.02	1	0x000000000001bd20	ld-linux-x86-64.s
1.78	0.02	1	_dl_catch_error	ld-linux-x86-64.s
1.73	0.01	1	0x000000000001b840	ld-linux-x86-64.s
1.54	0.07	1	0x0000000000019ef0	ld-linux-x86-64.s
1.49	0.05	1	0x00000000000065c0	ld-linux-x86-64.s
1.47	0.11	4	0x0000000000019e20	ld-linux-x86-64.s
1.38	0.52	1	0x0000000000018fc0	ld-linux-x86-64.s
1.27	0.02	1	_IO_doallocbuf	libc.so.6
1.26	0.04	1	_IO_file_doallocate	libc.so.6
1.18	0.04	1	malloc	libc.so.6
1.13	0.06	1	0x00000000000139a0	ld-linux-x86-64.s
1.07	0.26	14	0x000000000000ac70	ld-linux-x86-64.s

```
#   Ir    Source
 0        --- From '/home/tyler/Projects/CLanguageSnippets/build/../main.c' ---
 3        #include <stdint.h>
 4
 5        int main(int argc, char* argv[argc])
 6  0.00  {
 7  0.00      if (argc != 2)
 8         {
 9  0.01          printf("No argument found!\n");
    2.63  1 call(s) to '0x00000000000126a0' (ld-linux-x86-64.so.2)
10  0.00          return 1;
11         }
12         printf("%s\n", argv[1]);
13         return 0;
14  0.00  }
15
16
17
```

```
#   Ir   Hex   Assembly Instructions                       Source Position
1              There is no instruction info in the profile ...
2              Tip: For Callgrind, rerun with option
3                  --dump-instr=yes
4              To see (conditional) jumps, additionally s...
5                  --collect-jumps=yes
```

callgrind.out.174202 [1] - Total Instruction Fetch Cost: 147 742

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

1.2.6 PlantUML

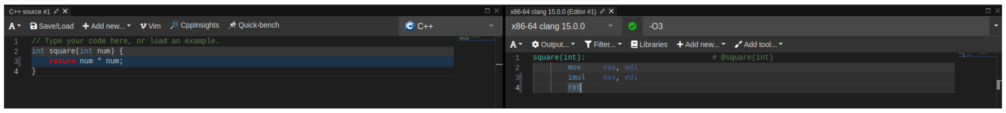
PlantUML is another utility that is used in Doxygen for generating diagrams by providing text input describing the diagram. This is a great tool for visualizing complex layouts and can prove invaluable for documentation.

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

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.



x86-64 clang 15.0.0 -O3

```
// Type your code here, or load an example.
int square(int num) {
    return num * num;
}
```

```
square(int):                            # @square(int)
        mov     eax, edi
        imul    eax, edi
        ret
```

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

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:

```
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:

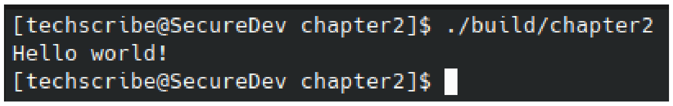
```c
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>

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:

```
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
```

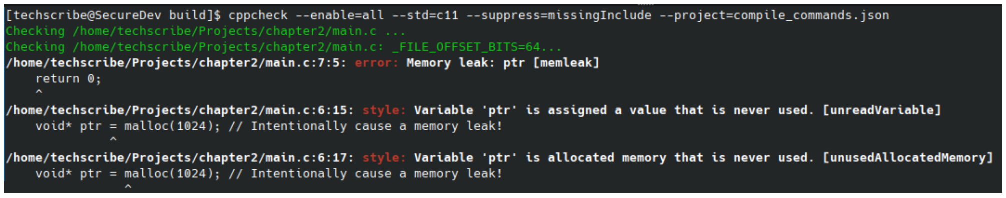
To see our cppcheck in action, let's change the code of our "main.c" file content to as followed:

```c
#include "stdint.h"
#include "stdlib.h"
#include "stdio.h"

int error() {
    void* ptr = malloc(1024); // Intentionally cause a memory leak!
    return 0;
}

int main(int argc, char* argv[])
{
    error();
    printf("Hello world! %s\n", argv[9]);
    return 0;
}
```

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 generates 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:

```c
#include "stdint.h"
#include "stdlib.h"
#include "stdio.h"

int error() {
    int* badPtr = NULL;
    return *badPtr;
}

int main(int argc, char* argv[])
{
    error();
    printf("Hello world! %s\n", argv[9]);
    return 0;
}
```

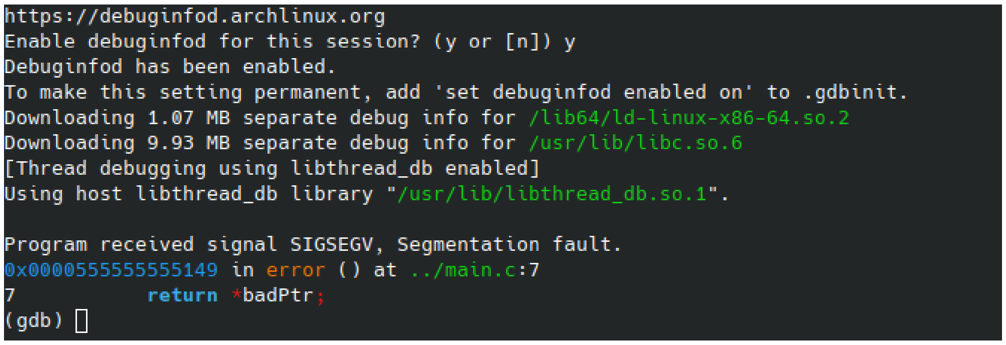
Now that we've updated our "main.c" file, we can go ahead and recompile the project. Make sure you have set the current directory to "chapter2" directory first:

```
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.
0x0000555555555149 in error () at ../main.c:7
7           return *badPtr;
(gdb) 
```

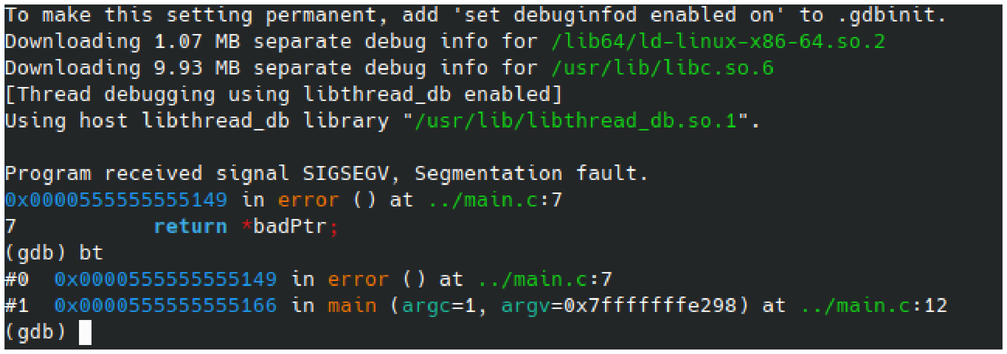
Figure 2.3: GDB Debugger Result

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.
0x0000555555555149 in error () at ../main.c:7
7           return *badPtr;
(gdb) bt
#0  0x0000555555555149 in error () at ../main.c:7
#1  0x0000555555555166 in main (argc=1, argv=0x7fffffffe298) 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

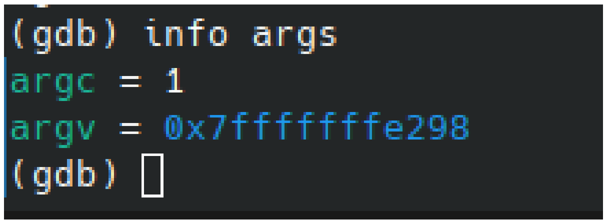
As mentioned before, we can walk up the stacktrace as we want, when you print a list of ”Frames”, you can see an index following after the # on the left hand side of each frame. We can select the frame within our main function by entering the following command:

```
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:

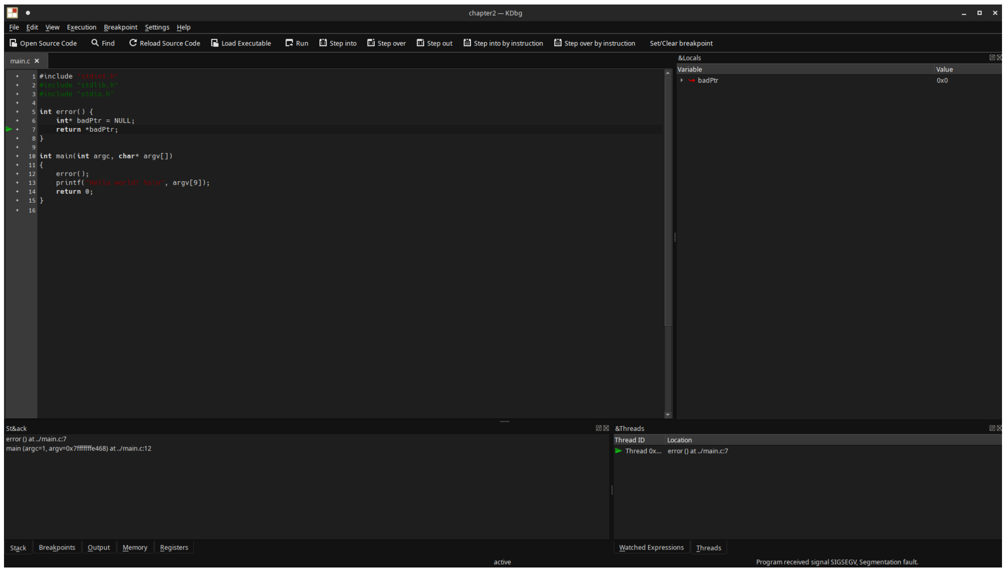


```
(gdb) info args
argc = 1
argv = 0x7fffffffe298
(gdb)
```

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.



```
 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 }
16
```

Variable	Value
badPtr	0x0

```
error () at ../main.c:7
main (argc=1, argv=0x7fffffffe468) at ../main.c:12
```

Thread ID	Location
Thread 0x...	error () at ../main.c:7

```
Program received signal SIGSEGV, Segmentation fault.
```

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 generates what is known as the Callgrind, which can be used to determines the hot paths in code. Enter the following code in "main.c" file content:

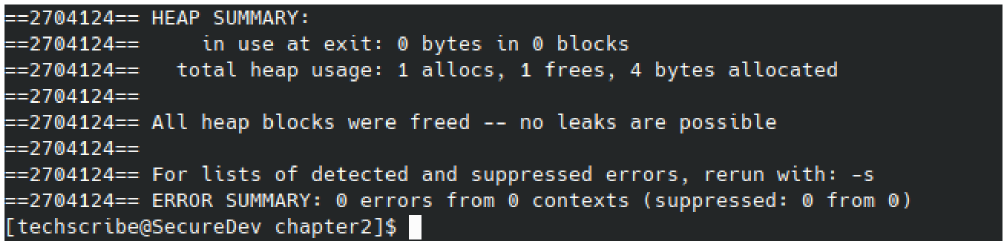
```c
#include "stdint.h"
#include "stdlib.h"
#include "stdio.h"

int* demo(int value)
{
    int* ptr = (int*)malloc(sizeof(int));
    *ptr = value;
    return ptr;
}
int main(int argc, char* argv[])
{
    int* ptr = demo(123);
    free(ptr);
    return 0;
}
```

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 reflects as followed:

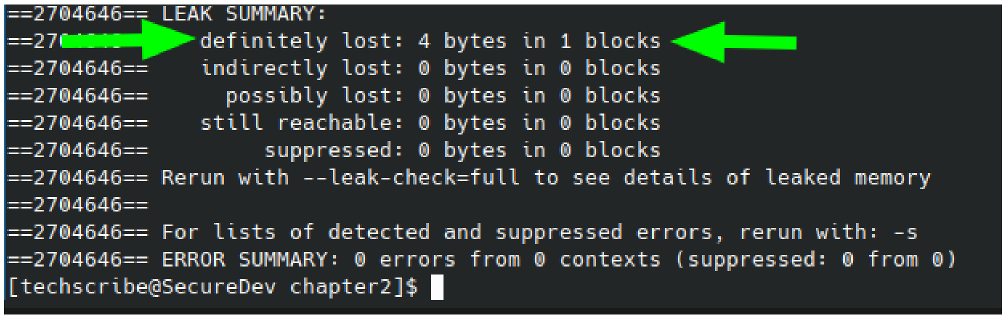
```c
#include "stdint.h"
#include "stdlib.h"
#include "stdio.h"

int* demo(int value)
{
    int* ptr = (int*)malloc(sizeof(int));
    *ptr = value;
    return ptr;
}
int main(int argc, char* argv[])
{
    int* ptr = demo(123);
    //free(ptr);
    return 0;
}
```

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:
==27           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==
==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 detects 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

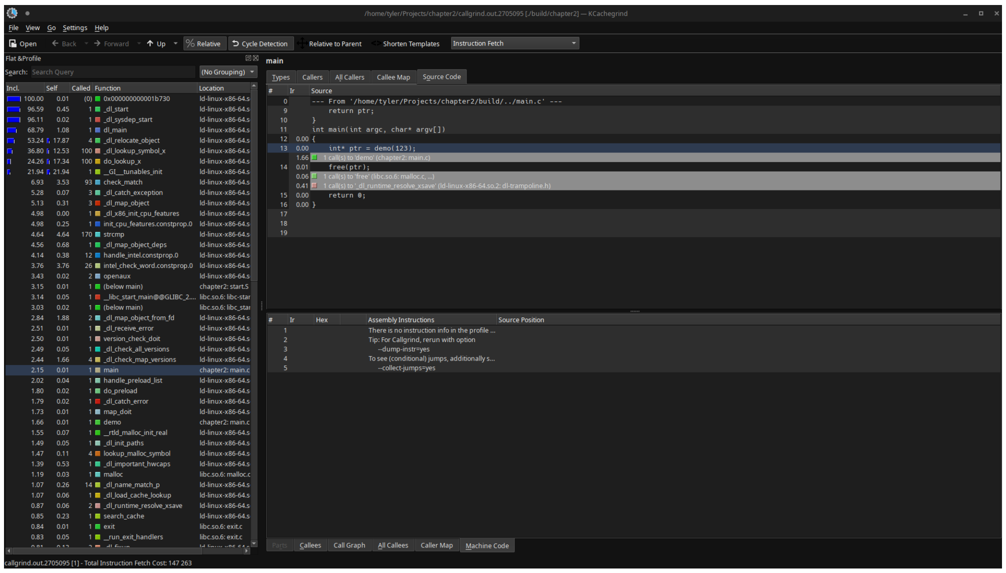
Another tool available to you from valgrind is call a Callgrind, as explained before, it allows us to detect hot paths in our code. First, let's generate a callgraph output file that we can then analyze later. Run the following commands in terminal:

```
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:



```
/home/tyler/Projects/chapter2/callgrind.out.2705095 [./build/chapter2] — KCachegrind
```

```
Incl.    Self    Called  Function                          Location
100.00   0.01    (0)     0x000000000001b730                ld-linux-x86-64.s
 96.59   0.45      1     _dl_start                         ld-linux-x86-64.s
 96.11   0.02      1     _dl_sysdep_start                  ld-linux-x86-64.s
 68.79   1.08      1     dl_main                           ld-linux-x86-64.s
 53.24  17.87      4     _dl_relocate_object               ld-linux-x86-64.s
 36.80  12.53    100     _dl_lookup_symbol_x               ld-linux-x86-64.s
 24.26  17.34    100     do_lookup_x                       ld-linux-x86-64.s
 21.94  21.94      1     __GI___tunables_init              ld-linux-x86-64.s
  6.93   3.53     93     check_match                       ld-linux-x86-64.s
  5.28   0.07      3     _dl_catch_exception               ld-linux-x86-64.s
  5.13   0.31      3     _dl_map_object                    ld-linux-x86-64.s
  4.98   0.00      1     _dl_x86_init_cpu_features         ld-linux-x86-64.s
  4.98   0.25      1     init_cpu_features.constprop.0     ld-linux-x86-64.s
  4.64   4.64    170     strcmp                            ld-linux-x86-64.s
  4.56   0.68      1     _dl_map_object_deps               ld-linux-x86-64.s
  4.14   0.38     12     handle_intel.constprop.0          ld-linux-x86-64.s
  3.76   3.76     26     intel_check_word.constprop.0      ld-linux-x86-64.s
  3.43   0.02      2     openaux                           ld-linux-x86-64.s
  3.15   0.01      1     (below main)                      chapter2: start.S
  3.14   0.05      1     __libc_start_main@@GLIBC_2....    libc.so.6: libc-star
  3.03   0.02      1     (below main)                      libc.so.6: libc_star
  2.84   1.88      2     _dl_map_object_from_fd            ld-linux-x86-64.s
  2.51   0.01      1     _dl_receive_error                 ld-linux-x86-64.s
  2.50   0.01      1     version_check_doit                ld-linux-x86-64.s
  2.49   0.05      1     _dl_check_all_versions            ld-linux-x86-64.s
  2.44   1.66      4     _dl_check_map_versions            ld-linux-x86-64.s
  2.15   0.01      1     main                              chapter2: main.c
  2.02   0.04      1     handle_preload_list               ld-linux-x86-64.s
  1.80   0.02      1     do_preload                        ld-linux-x86-64.s
  1.79   0.02      1     _dl_catch_error                   ld-linux-x86-64.s
  1.73   0.01      1     map_doit                          ld-linux-x86-64.s
  1.66   0.01      1     demo                              chapter2: main.c
  1.55   0.07      1     __rtld_malloc_init_real           ld-linux-x86-64.s
  1.49   0.05      1     _dl_init_paths                    ld-linux-x86-64.s
  1.47   0.11      4     lookup_malloc_symbol              ld-linux-x86-64.s
  1.39   0.53      1     _dl_important_hwcaps              ld-linux-x86-64.s
  1.19   0.03      1     malloc                            libc.so.6: malloc.c
  1.07   0.26     14     _dl_name_match_p                  ld-linux-x86-64.s
  1.07   0.06      1     _dl_load_cache_lookup             ld-linux-x86-64.s
  0.87   0.06      2     _dl_runtime_resolve_xsave         ld-linux-x86-64.s
  0.85   0.23      1     search_cache                      ld-linux-x86-64.s
  0.84   0.01      1     exit                              libc.so.6: exit.c
  0.83   0.05      1     __run_exit_handlers               libc.so.6: exit.c
```

```
#    Ir     Source
 0          --- From '/home/tyler/Projects/chapter2/build/../main.c' ---
 9              return ptr;
10          }
11          int main(int argc, char* argv[])
12   0.00   {
13   0.00       int* ptr = demo(123);
     1.66   1 call(s) to 'demo' (chapter2: main.c)
14   0.01       free(ptr);
     0.06   1 call(s) to 'free' (libc.so.6: malloc.c, ...)
     0.41   1 call(s) to '_dl_runtime_resolve_xsave' (ld-linux-x86-64.so.2: dl-trampoline.h)
15   0.00       return 0;
16   0.00   }
17
18
19
```

```
#   Ir   Hex   Assembly Instructions                          Source Position
1                There is no instruction info in the profile ...
2                Tip: For Callgrind, rerun with option
3                    --dump-instr=yes
4                To see (conditional) jumps, additionally s...
5                    --collect-jumps=yes
```

```
callgrind.out.2705095 [1] - Total Instruction Fetch Cost: 147 263
```

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.

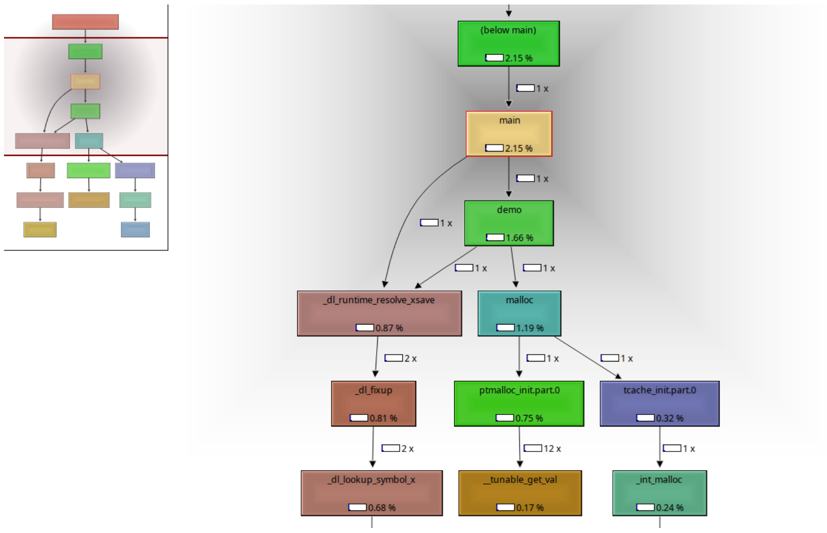


Figure 2.10: The Callgraph

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 generates 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:

```c
#include "stdint.h"
#include "stdlib.h"
#include "stdio.h"

/// @brief A structure that represents the information
///        about a human being, its name and age.
typedef struct Human {
    /// @brief Name of a human, should not exceed 512 characters.
    char name[512];
    /// @brief Age of a human represented in number of years
    ///        after birth, age value cannot be below 0.
    uint age;
} Human;

int main(int argc, char* argv[])
{
    Human TechScribe = (Human){
        .age = 50,
        .name = "TechScribe"
    };

    printf("This human being name is %s and it is %u years old.\n",
           TechScribe.name, TechScribe.age);
    return 0;
}
```

As you may have noticed the new comment lines following triple backslashes and "@brief", it is used to let Doxygen knows 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 it's fields annonated.

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
```

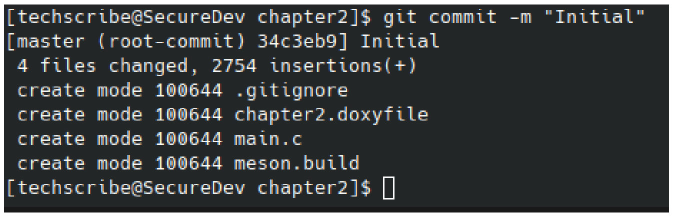
One of the first thing you may want to make before committing your files to git repository is to create a git ignore file which is used to list all of the files or directories you wish to be excluded from git history. Create a new ".gitignore" file, and make sure that the period in front of gitignore 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/.

CHAPTER 3

INTRODUCTION TO C PROGRAMMING

Now that we have covered the many tools that you'll be using for C Programming, let's first clone a project that I've prepared for your learning path, the BigProject template:

```
cd ~/Projects
git clone https://github.com/TechScribe-Deaf/BigProject
cd BigProject
git submodule update --init --recursive
```

With our new BigProject set up, we're finally ready to start learning C Programming!

VARIABLES

Variable in C is a named memory that holds a value, it could be a number, text, address, or anything else. The variables have to be described by using Type and type in C language describe what kind of memory it is and the shape it will have, for instance, a type int32_t describes an integer that holds 32 bits of memory ranging between -2147483648 to 2147483647 without representing any decimal point. And each byte represents 8 bits (usually unless you're running on a very old machine), so an integer results in 4 bytes being occupied. Open "main.c" under " /Projects/BigProject/src/" directory and write the followings in main function:

```c
int main(int argc, char* argv[argc])
{
    int32_t variable = 3;
    return variable;
}
```

With our variable successfully created, we holds 4 bytes of memory within main function, but once the function ends, that memory is then immediately released. This is called the stack allocated memory which happens in the function call stack, because you don't have to free the memory allocated here. This is something we describe as temporary memory allocation.

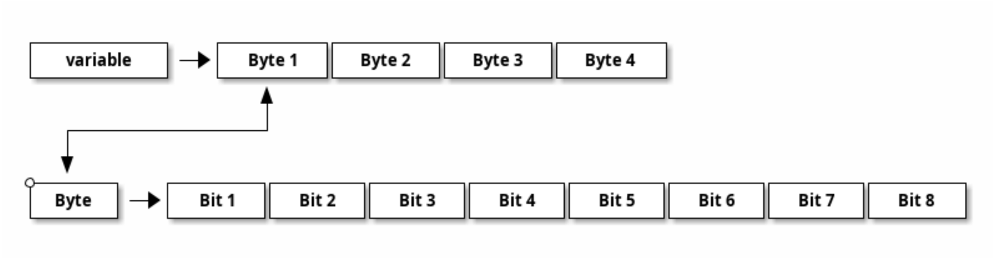


Figure 3.1: Variable Memory Layout

You can also allocate a number of variables on a stack by creating an array as well by writing the following:

```c
int main(int argc, char* argv[argc])
{
    int32_t variable[4] = {2};
    variable[1] = 3;
    return variable[0];
}
```

When initializing a variable with "= {2}", it let compiler know that you want to set the first integer value to 2 and any other unspecified value for other indexes would default to 0. The variable will looks like the following memory diagram after running the main function:

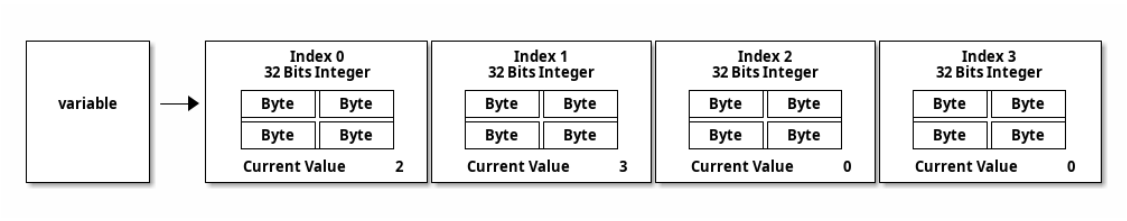


Figure 3.2: Variables Memory Layout

You can retrieve the value by using square brackets ("[...]") to look up for value within a variable that contains a number of values by providing an index. If you check the diagram, you'll notice that when main function terminates, a value of 2 will be returned.

TYPES

A type is how we describe a blob of memory of what shape it has and how it should be used in our program. Is something an integer? If so, how big is that integer? What rules should be applied to that integer if we want to add and subtract? For instance, if we have two different kinds of integer, one integer have a sign where it can represents both positive and negative number, and the other one is an unsigned integer which can only represents positive number. There are different rules associated for each type of integer, so let's first understands the difference between signed integer and unsigned integer.

3.2.1 Unsigned and Signed Integer

A signed integer have a range bteween 2,147,483,647 to -2,147,483,648, when you starts with 0 as an integer and then subtract it by 1, you would get -1 for answer. An unsigned integer have a range of 0 to 4,294,967,295, and so when you starts with 0 as an unsigned integer and then subtracts it by 1, you would get 4,294,967,295 for answer. The reason for that is called an underflow when we are trying to represents a number below it's permissible range of number that it can represents, it rolls around to it's highest number and vice versa for overflow where number rolls around from it's highest value to it's lowest value. The same holds true for signed integer if you for instance try to subtract -2,147,483,648 by 1 and you would see 2,147,483,647 for answer.

```c
int main(int argc, char* argv[argc])
{
    uint32_t unsignedNumber = 0;
    int32_t signedNumber = 0;
    printf("%u\n", unsignedNumber - 1); // 4294967295 (This is an underflow!)
    printf("%i\n", signedNumber - 1);   // -1
    signedNumber =  ;
    unsignedNumber = 4294967295;
    printf("%u\n", unsignedNumber + 1); // 0 (This is an overflow!)
    printf("%i\n", signedNumber + 1);   // -2147483648 (This is an overflow!)
    return 0;
}
```

To summarize:

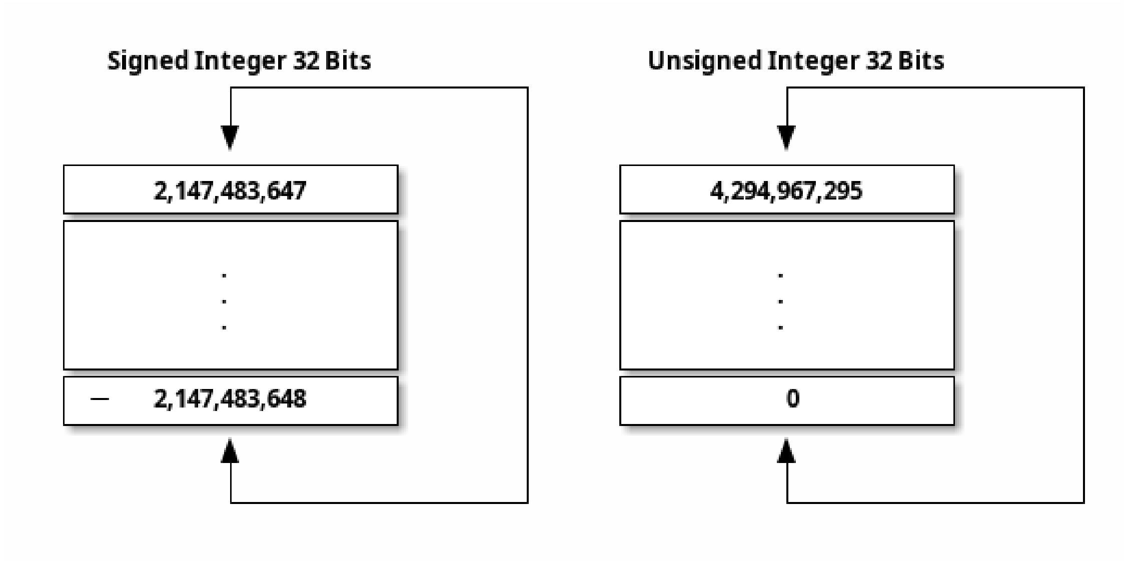


Figure 3.3: Underflow and Overflow Diagram

3.2.2 Ambiguity of Integer Sizes

C Programming Language offers some ambiguity on just how big an integer really is when you use ”int”, because according to C Language specification, it only indicates that ”int” is *at least* capable of representing -32768 to 32767 or 2 bytes sized integer, but it is commonly instead sized up to -2147483648 to 2147483647 or 4 bytes integer in most desktops today. If you want to rule out ambiguity sized integers, then you would need to include a header file which is already in your ”main.c” file in the BigProject, ’#include ”stdint.h”’ which is used to defines variety of types giving you exact sized integers to use within your application. It also allows you to rule out a class of bugs by taking confusion out of the equation when writing your program. There are a number of types provided such as ”int8_t”, ”int16_t”, ”int32_t”, ”int64_t” and it’s unsigned components, ”uint8_t”, ..., ”uint64_t”. For the rest of this book, we will only use ambigious integer types for main function to conforms to the norm, but we will be using sized integers for everything else as this is a practice encouraged within this book.

3.2.3 Structures

Structure in C allows us to create a number of types within a single type for convinence and preventing a class of bugs seen in assembly programming. In C, to declare a structure, we must first use keyword, "struct" and then give this structure an unique name and then define it's body surrouned in curly brackets of various members with it's type and name defined. Each member should be terminated with semicolon instead of commas.

```c
struct MyStruct {
    int32_t FirstMember;
    uint32_t SecondMember;
};
```

The data structure would looks like the following diagram, this structure essentially packs 2 integers with clearly defined boundaries between members.

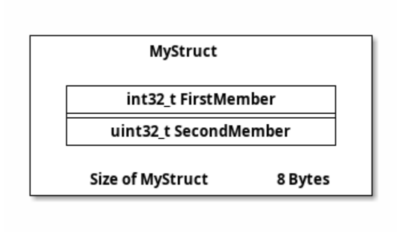


Figure 3.4: MyStruct Diagram

To declare a variable with a type of a defined structure, you would need to use keyword struct first to indicates that compiler should look for all types that are declared as a structure. After struct keyword, you specify the name of MyStruct, and then enter the name of variable and then you can close it with a semicolon.

```c
struct MyStruct myvariable;
```

It should be noted that when using defined structure variable above without initializing it and you still use any of it's members, this is something we called an undefined behavior, because from this point on, we cannot guarantee what value you're going to get from that structure, it may be zero, or it may a random value. You can try this out for yourself:

```c
struct MyStruct {
    int32_t FirstMember;
    uint32_t SecondMember;
};

int main(int argc, char* argv[argc])
{
    struct MyStruct myvar;
    printf("%i\n", myvar.FirstMember);
    return 0;
}
```

3.2.4 Initializing Struct

There are generally two approaches you can use to initialize your structure, the first approach is to assign the uninitialized structure variable to an object surrounded by curly brackets with each member name prefixed by a period assigned to value and have each member separated by commas within curly brackets and then add semicolon after the curly brackets, or to simply access each structure member to a value or variable by specifying the structure variable and then add a period and then specify member name and then have it assign to a value.

```c
int main(int argc, char* argv[argc])
{
    struct MyStruct myvar = { .FirstMember = 123, .SecondMember = 321 };
    printf("%i\n", myvar.FirstMember);
    return 0;
}
```

OR

```c
int main(int argc, char* argv[argc])
{
    struct MyStruct myvar;
    myvar.FirstMember = 123;
    myvar.SecondMember = 321;
    printf("%i\n", myvar.FirstMember);
    return 0;
}
```

It should be noted that it is not required to assign every member in struct to a value when using curly brackets initialization for a struct variable. The following example is a perfectly valid code.

```c
int main(int argc, char* argv[argc])
{
    struct MyStruct myvar = {};
    printf("%i\n", myvar.FirstMember);
    return 0;
}
```

FUNCTIONS

A function in C language have 4 parts, a return type, name of function, parameter type, parameter name.

To declare a function (Function Declaration) first a return type describes what kind of data is being returned after running the function, would it returns an integer, text, or block of data, how big would it be, and more. Next, name of a function should be unique, and distinct from any previously defined function. If you are designing a library, it's usually a good idea to make a short name prefix for your functions like for instance, "libtest_add" or "libgraphic_paint". A function have any number of parameters, it could be zero parameter and potentially thousands parameter if desired, and each parameter should be defined by it's type and it's unique parameter name, and all parameter should be enclosed in parathesis brackets.

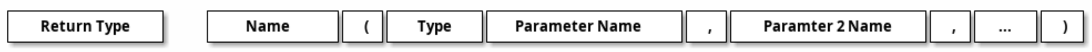


Figure 3.5: Function Declaration Diagram

The body of the function is optional, you can end this with a semicolon to let C Programming Language compiler knows that this function *will* be available or defined later, but to allow it to be used within your code for now, or you can define a body of the function by surrounding it in curly brackets. When you write a function with a return type of ”void”, it does not require you to return any value, but if you specify it to anything else, then you are required to return a value prior to terminating the function.

```c
void Returns_Nothing() {
    // Not required to return anything! No value can be obtained!
}
int32_t Returns_An_Integer() {
    return 1;
}
```

OPERATORS

There are many types of operators we can use in C Language, but arithmetic operators can only be apply to Integer or Floating Point. Floating point is another kind of number that represents a decimal point and can only approximate accuracy depending on the memory size of this number. Here are the following arithmetic operators that are available in C:

3.4.1 Arithmetic Operators

Name of Operator	Description	Operator	Example
Addition	Adds The Two Values	+	11 + 2 = 13
Subtraction	Subtracts The Two Values	-	11 - 2 = 9
Multiply	Multiplies The Two Values	*	11 * 2 = 22
Divide	Divides The Two Values	/	11 / 2 = 5
Modulus	Get Remainder After Division Of The Two Values	%	11 % 2 = 1

Example:

```c
int main(int argc, char* argv[argc])
{
    int32_t variable = 1;
    variable = variable + 2; // 1 + 2 = 3
    variable = variable - 4; // 3 - 4 = -1
    variable = variable * 20; // -1 * 20 = -20
    variable = variable / 3; // -20 / 3 = -6
    variable = variable % 2; // -6 % 2 = 0
    printf("%i\n", variable); // Will prints 0
    return 0;
}
```

3.4.2 Comparison Operators

As for comparison operators, it will be important to touch upon the basic operators listed below as we will be evaluating conditional logic in our program in the next step.

Name of Operator	Description	Operator	Examples
Equality Comparison	Checks if two values are equals or not. If values are equal, the resulting comparison becomes 1, otherwise 0 if the two values are not equals.	==	1 == 1 = 1 2 == 1 = 0 0 == 0 = 1
Inequality Comparison	Checks if two values are not equals. If values are not equal, the resulting comparison becomes 1, otherwise 0 if the two values are equals.	!=	1 != 1 = 0 2 != 1 = 1 0 != 0 = 0
More Than Comparison	Checks if the value on the left is more than the value on the right. If value on the left is greater than the right value, the resulting comparison becomes 1, otherwise 0.	>	1 > 1 = 0 2 > 1 = 1 0 > 0 = 0
Less Than Comparison	Checks if the value on the left is less than the value on the right. If value on the left is lesser than the value on the right, the resulting comparison becomes 1, otherwise 0.	<	1 < 1 = 0 1 > 2 = 1 0 < 0 = 0
More Than Or Equals To Comparison	Checks if the value on the left is greater than or equals to the value on the right, the resulting comparison becomes 1, otherwise 0.	>=	1 >= 1 = 1 2 >= 1 = 1 0 >= 1 = 0
Less Than Or Equals To Comparison	Checks if the value on the left is lesser than or equals to the value on the right, the resulting comparison becomes 1, otherwise 0.	<=	1 <= 1 = 1 2 <= 1 = 0 0 <= 1 = 1

Example:

```c
int main(int argc, char* argv[argc])
{
    int32_t l_value = 5;
    int32_t r_value = 10;
    printf("%i == %i = %i\n", // Prints "5 == 10 = 0"
            l_value, r_value, l_value == r_value);
    printf("%i != %i = %i\n", // Prints "5 != 10 = 1"
            l_value, r_value, l_value != r_value);
    printf("%i > %i = %i\n",  // Prints "5 > 10 = 0"
            l_value, r_value, l_value > r_value);
    printf("%i < %i = %i\n",  // Prints "5 < 10 = 1"
            l_value, r_value, l_value < r_value);
    printf("%i >= %i = %i\n", // Prints "5 >= 10 = 0"
            l_value, r_value, l_value >= r_value);
    printf("%i <= %i = %i\n", // Prints "5 <= 10 = 1"
            l_value, r_value, l_value <= r_value);
    return 0;
}
```

3.4.3 Logical Operators

Name of Operator	Description	Operator	Examples
Lazy AND Operator	Evaluates whether both values return non-zero value, if the value on the left returns a zero, then value on the right may not be evaluated. If both values are non-zero, then this returns 1, otherwise 0.	&&	1 && 0 = 0 1 && 1 = 1 0 && 0 = 0
Lazy OR Operator	Evaluates whether either values return non-zero value, if the value on the left returns a non-zero, then value on the right may not be evaluated. If either value is non-zero, then this returns 1, otherwise 0.	||	1 || 0 = 1 1 || 1 = 1 0 || 0 = 0
Logical NOT Operator	NOT Operator reverse the logical state of a given value, if the value is non-zero, then it will be set to zero, otherwise it will be set to 1.	!	!0 = 1 !1 = 0

Example:

```c
int32_t variable = 0;

int true()
{
    variable = variable + 1;
    return 1;
}

int false()
{
    variable = variable + 1;
    return 0;
}

/// @brief The first function to be executed.
int main(int argc, char* argv[argc])
{
    printf("true() && false() = %i, variable was added %i times\n",
           true() && false(), variable);
    variable = 0; // variable resets to zero.
    printf("true() || false() = %i, variable was added %i times\n",
           true() || false(), variable);
    variable = 0; // variable resets to zero.
    printf("false() || true() = %i, variable was added %i times\n",
           false() || true(), variable);
    printf("!1 = %i and !0 = %i and !5 = %i and !-5 = %i\n",
           !1, !0, !5, !-5);
    return 0;
}
/* Output prints as followed:
true() && false() = 0, variable was added 2 times
true() || false() = 1, variable was added 1 times
false() || true() = 1, variable was added 2 times
!1 = 0 and !0 = 1 and !5 = 0 and !-5 = 0
*/
```

3.4.4 Shorthand Operators

There are a number of assignment operators as you may have seen before, it allows you to assign a variable to a certain value. Other assignment operators also allows you to make a shorthand operations while assigning the value to resulting value.

Operator	Description	Shorthand For
=	Assign the variable on the left to the value on the right.	
+=	Adds with the variable on the left with the value on the right and then assign to the variable.	Y = Y + X
-=	Subtracts with the variable on the left with the value on the right and then assign to the variable.	Y = Y - X
*=	Multiplies with the variable on the left with the value on the right and then assign to the variable.	Y = Y * X
/=	Divides with the variable on the left with the value on the right and then assign to the variable.	Y = Y / X
%=	Modulus with the variable on the left with the value on the right and then assign to the variable.	Y = Y % X

```c
int main(int argc, char* argv[argc])
{
    int32_t variable = 0;
    variable += 2;
    printf("%i\n", variable); // 2
    variable -= 3;
    printf("%i\n", variable); // -1
    variable *= 20;
    printf("%i\n", variable); // -20
    variable /= 3;
    printf("%i\n", variable); // -6
    variable %= 2;
    printf("%i\n", variable); // 0
    return 0;
}
```

BASIC FUNCTIONS

As we will be covering the function concepts later, it's important to first familiarize ourselves with some few basic functions we'll need for our project.

3.5.1 Printf Function

Printf is a function provided by ”stdio.h” header file, and it allows us to print a message onto the terminal. We can create a string enclosed in double quotes and display a message like so:

```c
#include "stdio.h"
#include "stdint.h"

int main(int argc, char* argv[argc])
{
    printf("Hello world!\n");
    int32_t number = 123;
    printf("I can print number by doing this: %i\n", number);
    return 0;
}
```

As you may see in the second printf function call, we are able to specify printf to print our number by providing the formatter specifiers which describes the type of parameter it is expecting to print for, you can also customize on how it prints that value too. Wikipedia offers a good reference for printf format string which you can find at https://en.wikipedia.org/wiki/Printf_format_string.

And you may have noticed ”\n” at the end of the string as well, and this is something that is known as the escape sequences and ”\n” is a code for adding a new line to the string being printed in terminal.

Escape Sequence	Description
\n	New Line, add a new line after a given text.
\t	Horizontal tab, shift the following text by preset space for tab.
\b	Backspace, delete a letter before this escape sequence.
\r	Carriage return, reset the position to the beginning of a line of text.
\a	Sounds a bell, it will generates a sound of a bell. *(You may have to edit Terminal Emulator profile to use system bell to generates sound.)*
\́	Prints single quote.
\̈	Prints double quotes.
\?	Prints question mark.
\\	Prints back slash.
\f	Advances the text by a page.
\v	Vertical tab, similarly to inserting newline and then continuing where it's left off horizontally.
\0	Null value, this is commonly used to terminates text/string when C Program read a given string.