



FIRST EDITION – CHAPTER 4 REV 1

Kevin Thomas  
Copyright © 2022 My Techno Talent

# Forward

I remember a time before the days of the internet where computers were simple yet elegant and beautiful in their design, logic and functionality.

I was a teenager in the 1980's when I got my first Commodore 64 for Christmas and the first thing I did was tear it out of the box and get it wired up to my console TV as the only thing I needed to see was that blinking console cursor on that blue background with the light blue border.

It was a blank slate. There were no libraries. There were no frameworks. If you wanted to develop something outside of the handful of games that you could get for it, you program it from scratch.

In addition to the C-64 there was a 300 baud modem with a 5.25" floppy disk which read DMBBS 4.8. I quickly read the small documentation that came with it and quickly took over the only phone line in the household.

I set up my BBS or bulletin board system, and called it THE ALLNIGHTER. I set it up and no one called obviously as no one knew it existed. I joined a local CUF group, computer user federation, where I met another DMBBS 4.8 user which helped me network my message boards to him.

At a given time of day my computer would call his and send my messages to his board and I would receive his messages from his board. It was computer networking before the internet and it was simply magic.

Over the next few months he taught me 6502 Assembler which was my first programming language that I ever learned. Every single instruction was given consideration of the hardware and a mastery over the computer was developed as we did literally everything from scratch on the bare metal of the hardware.

Today we live in an environment of large distributed systems where there are thousands of libraries and dozens of containers within pods in a large orchestrated Kubernetes cluster which defines an application.

Between the 1980's and current, the birth of higher-level languages has made it possible to develop in a timely manner even on the most sophisticated distributed systems.

As we work within a series of large cloud ecosystems, there exists a programming language called Golang, or Go for short, which allows for easy software development to take advantage of multiple cores within a modern CPU in addition to out-of-the-box currency and ease of scale for enterprise-level network and product design.

With every great technology there arises threat actors that exploit such power.

Go can be compiled easily for multiple operating systems producing a single binary. The speed and power of Go makes it an easy choice for modern Malware Developers.

There are literally thousands of books and videos on how to reverse engineer traditional C binaries but little on Go as it is so relatively new.

The aim of this book is to teach basic Go and step-by-step reverse engineer each simple binary to understand what is going on under the hood.

We will develop within the Windows architecture (Intel x64 CISC) as most malware targets this platform by orders of magnitude.

Let's begin...

# Table Of Contents

Chapter 1: Hello Distributed System World  
Chapter 2: Debugging Hello Distributed System World  
Chapter 3: Hacking Hello Distributed System World  
Chapter 4: Primitive Types

# Chapter 1: Hello Distributed System World

We begin our journey with developing a simple hello world program in Go on a Windows 64-bit OS.

We will then reverse engineer the binary in IDA Free.

Let's first download Go for Windows.

<https://go.dev/doc/install>

Let's download IDA Free.

<https://hex-rays.com/ida-free/#download>

Let's download Visual Studio Code which we will use as our integrated development environment.

<https://code.visualstudio.com/>

Once installed, let's add the Go extension within VS Code.

<https://marketplace.visualstudio.com/items?itemName=golang.go>

Let's create a new project and get started by following the below steps.

New File  
main.go

Now let's populate our **main.go** file with the following.

```
package main

import "fmt"

func main() {
    fmt.Println("hello distributed system world")
}
```

Let's open up the terminal by click CTRL+SHIFT+` and type the following.

```
go mod init main
go mod tidy
go build
Let's run the binary!
```

.\main.exe

Output...

hello distributed system world

Congratulations! You just created your first hello world code in Go.  
Time for cake!

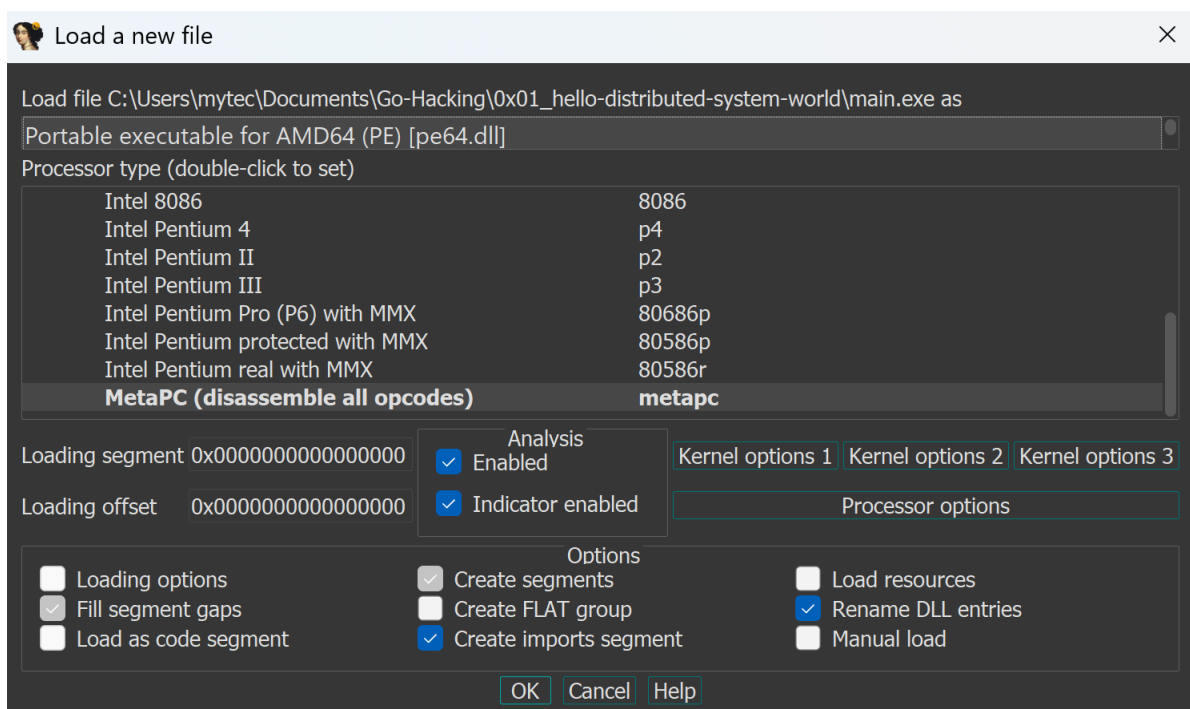
We simply created a hello world style example to get us started.

In our next lesson we will debug this in IDA Free!

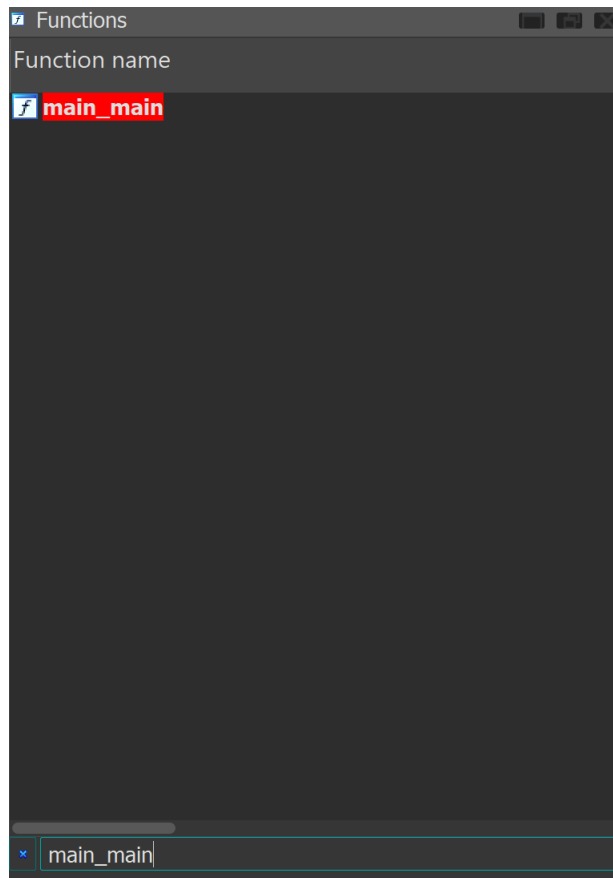
## Chapter 2: Debugging Hello Distributed System World

Let's debug our app within IDA Free.

Open IDA Free and we see the load screen. We can keep all the defaults and simply click OK.

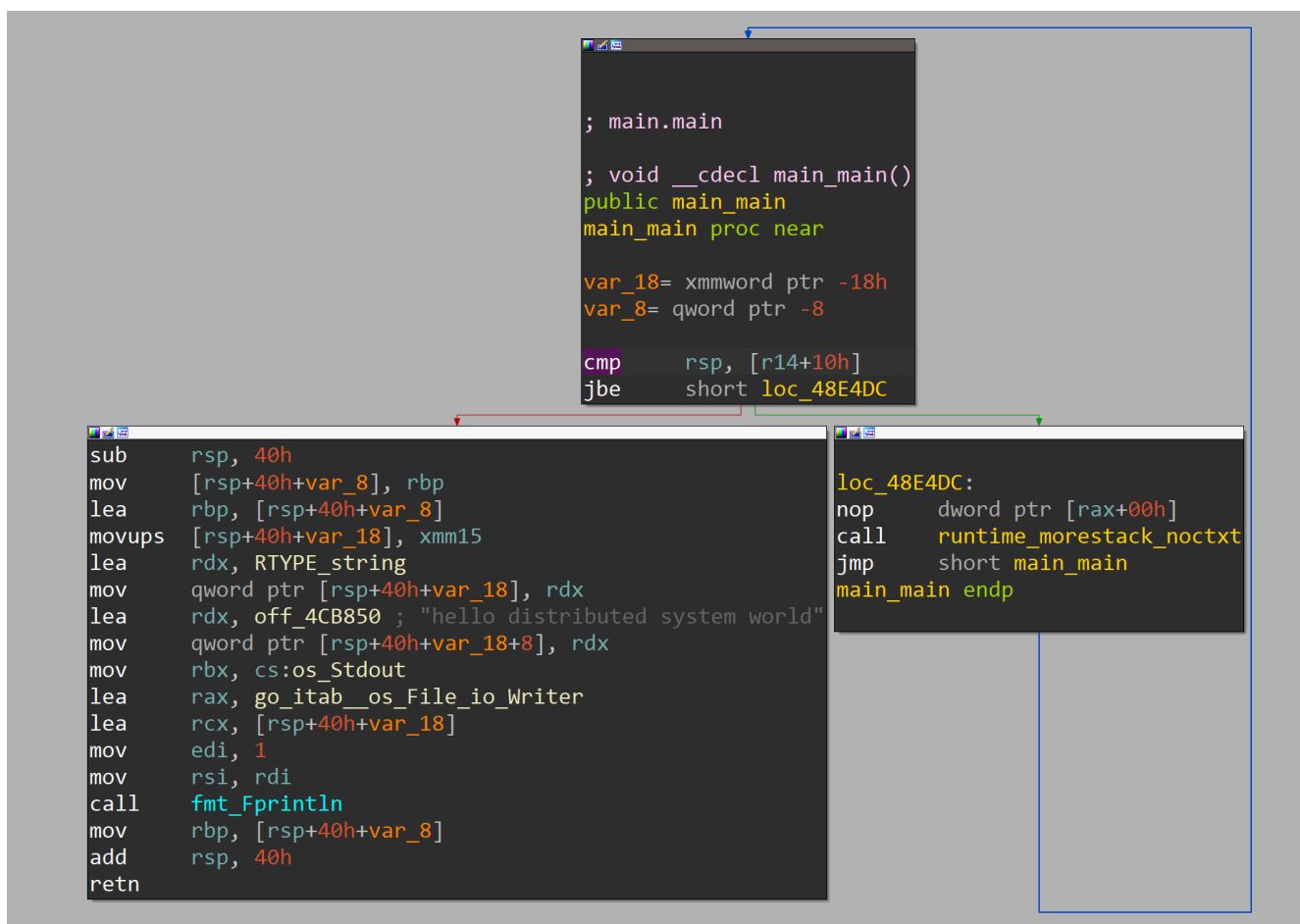


In Go at the assembler level we will need to search for the entry point of our app. This is the *main\_main* function. You can use CTRL+F to search.



Now we can double-click on the *main\_main* to launch the focus to this function and graph.





We can see in the bottom left box our *“hello distributed system world”* text.

If we double-click on *off\_4CB850* it will take us to a new window where the string lives within the binary.

```

• .rdata:00000000004CB850 off_4CB850      dq offset aHelloDistribut
  .rdata:00000000004CB850                                ; DATA XREF: main_main+26↑to
  .rdata:00000000004CB850                                ; "hello distributed system world"
• .rdata:00000000004CB858                        db  1Eh

```

Here we see something very interesting. Unlike a C binary where the string is terminated by a null character, we see that there is the raw string in a large pool and a *1eh* value which represents the length of the string in hex.

If we double-click on the *“hello distributed system world”* text we will see the string pool within the binary.

```

• .rdata:00000000004ADCFD aFreedeferWithD_0 db 'freedefer with d._panic != nil'
.rdata:00000000004ADCFD ; DATA XREF: runtime_freedeferpanic+14↑to
• .rdata:00000000004ADD1B aHelloDistribut db 'hello distributed system world'
.rdata:00000000004ADD1B ; DATA XREF: .rdata:off_4CB850↓to
• .rdata:00000000004ADD39 aInappropriateI db 'inappropriate ioctl for device'
.rdata:00000000004ADD39 ; DATA XREF: .data:0000000000541690↓to
• .rdata:00000000004ADD57 aInvalidPointer db 'invalid pointer found on stack'
.rdata:00000000004ADD57 ; DATA XREF: runtime_adjustpointers+1BF↑to
• .rdata:00000000004ADD75 aNotetsleepWait db 'notetsleep - waitm out of sync'
.rdata:00000000004ADD75 ; DATA XREF: runtime_notetsleep_internal:loc_40A906↑to
• .rdata:00000000004ADD93 aProtocolWrongI db 'protocol wrong type for socket'
.rdata:00000000004ADD93 ; DATA XREF: .data:0000000000541740↓to
• .rdata:00000000004ADDB1 aReflectElemOfI_0 db 'reflect: Elem of invalid type '
.rdata:00000000004ADDB1 ; DATA XREF: reflect_ptr_rtype_Elem+112↑to
• .rdata:00000000004ADDCF aReflectLenOfNo db 'reflect: Len of non-array type'
.rdata:00000000004ADDCF ; DATA XREF: .rdata:off_4CB840↓to
• .rdata:00000000004ADDED aRunqputsLowQue db 'runqputslow: queue is not full'
.rdata:00000000004ADDED ; DATA XREF: runtime_runqputslow:loc_44353A↑to
• .rdata:00000000004ADE0B aRuntimeBadGInC db 'runtime: bad g in cgocallback',0Ah
.rdata:00000000004ADE0B ; DATA XREF: runtime_cgocallbackg+45↑to
• .rdata:00000000004ADE29 aRuntimeBadPoin db 'runtime: bad pointer in frame '
.rdata:00000000004ADE29 ; DATA XREF: runtime_adjustpointers+15F↑to
• .rdata:00000000004ADE47 aRuntimeFoundIn db 'runtime: found in object at *('
.rdata:00000000004ADE47 ; DATA XREF: runtime_badPointer+155↑to
• .rdata:00000000004ADE65 aRuntimeImpossi_0 db 'runtime: impossible type kind '
.rdata:00000000004ADE65 ; DATA XREF: runtime_typesEqual+C25↑to
• .rdata:00000000004ADE83 aSocketOperatio db 'socket operation on non-socket'
.rdata:00000000004ADE83 ; DATA XREF: .data:0000000000541670↓to

```

All of the strings are within this string pool which is a very different architectural design compared to other languages.

With this basic analysis we have a general idea of what is going on within this simple binary.

These lessons are designed to be short and digestible so that you can code and hack along.

In our next lesson we will learn how to hack this string and force the binary to print something else to the terminal of our choosing.

This will give us the first taste on hacking Go!

# Chapter 3: Hacking Hello Distributed System World

Let's hack our app within IDA Free.

In our last lesson we saw our large string pool. Lets load up IDA and revisit that pool.

```
.rdata:00000000004ADCFD aFreedeferWithD_0 db 'freedefer with d._panic != nil'
.rdata:00000000004ADCFD ; DATA XREF: runtime_freedeferpanic+14f0
.rdata:00000000004ADD1B aHelloDistribut db 'hello distributed system world'
.rdata:00000000004ADD1B ; DATA XREF: .rdata:off_4CB850lo
.rdata:00000000004ADD39 aInappropriateI db 'inappropriate ioctl for device'
.rdata:00000000004ADD39 ; DATA XREF: .data:0000000000541690lo
.rdata:00000000004ADD57 aInvalidPointer db 'invalid pointer found on stack'
.rdata:00000000004ADD57 ; DATA XREF: runtime_adjustpointers+1BFf0
.rdata:00000000004ADD75 aNotetsleepWait db 'notetsleep - waitm out of sync'
.rdata:00000000004ADD75 ; DATA XREF: runtime_notetsleep_internal:loc_40A906f0
.rdata:00000000004ADD93 aProtocolWrongT db 'protocol wrong type for socket'
.rdata:00000000004ADD93 ; DATA XREF: .data:0000000000541740lo
.rdata:00000000004ADDB1 aReflectElemOfI_0 db 'reflect: Elem of invalid type '
.rdata:00000000004ADDB1 ; DATA XREF: reflect_ptr_rtype_Elem+112f0
```

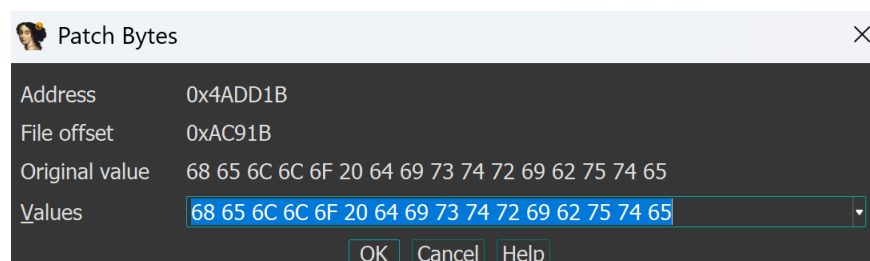
Let's select *Windows* then *Hex View-1*.

00000004ADCE0	61 69 6C 65 64 20 74 6F	20 67 65 74 20 73 79 73	ailed.to.get.sys
00000004ADCF0	74 65 6D 20 70 61 67 65	20 73 69 7A 65 66 72 65	tem.page.sizefre
00000004ADD00	65 64 65 66 65 72 20 77	69 74 68 20 64 2E 5F 70	edefer.with.d._p
00000004ADD10	61 6E 69 63 20 21 3D 20	6E 69 6C 68 65 6C 6C 6F	anic.!=.nilhello
00000004ADD20	20 64 69 73 74 72 69 62	75 74 65 64 20 73 79 73	.distributed.sys
00000004ADD30	74 65 6D 20 77 6F 72 6C	64 69 6E 61 70 70 72 6F	tem.worldinappro
00000004ADD40	70 72 69 61 74 65 20 69	6F 63 74 6C 20 66 6F 72	priate.ioctl.for

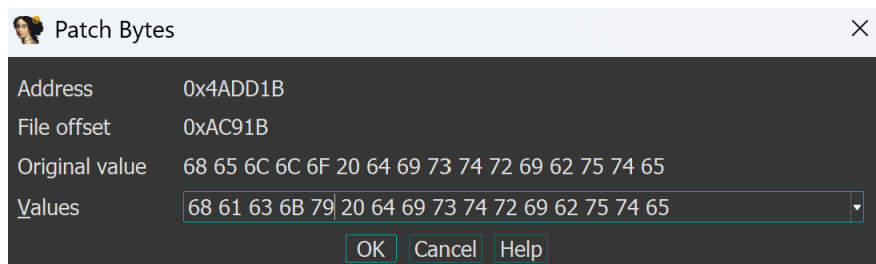
Here we see our string's hex values. It is literally as simple as this as this will be a very short and rewarding chapter.

Select *Edit*, *Patch program* then *Change byte...*

We can use the Ascii Table at <https://www.asciitable.com/> to change our string from, *hello distributed system world* to *hacky distributed system world* by simply patching the bytes



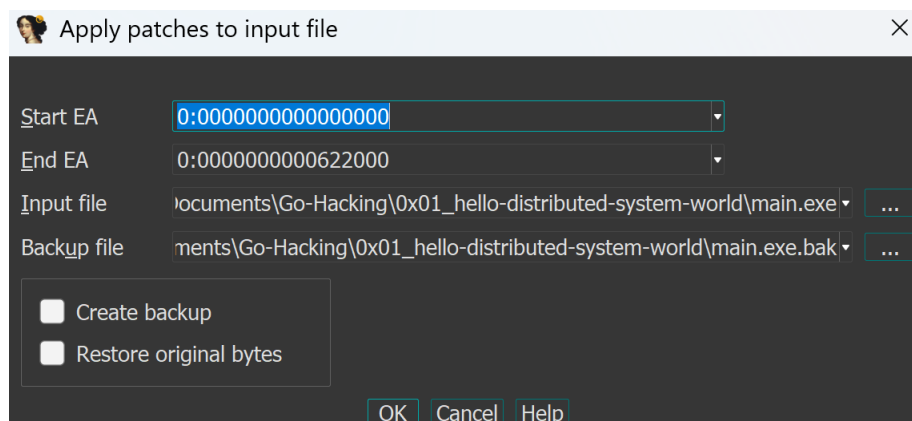
After we change hello to hacky we have the following.



Now we observe the following change.




































```
00000004ADD00 65 64 65 66 65 72 20 77 69 74 68 20 64 2E 5F 70 edefer.with.d._p
00000004ADD10 61 6E 69 63 20 21 3D 20 6E 69 6C 68 61 63 6B 79 anic.!=.nilhacky
00000004ADD20 20 64 69 73 74 72 69 62 75 74 65 64 20 73 79 73 .distributed.sys
00000004ADD30 74 65 6D 20 77 6F 72 6C 64 69 6E 61 70 70 72 6F tem.worldinappro
00000004ADD40 70 72 69 61 74 65 20 69 6F 63 74 6C 20 66 6F 72 priate.ioctl.for
00000004ADD50 20 64 65 76 69 63 65 69 6E 76 61 6C 69 64 20 70 .deviceinvalid.p
00000004ADD60 6F 69 6E 74 65 72 20 66 6F 75 6E 64 20 6F 6E 20 ointer.found.on.
00000004ADD70 73 74 61 63 6B 6E 6F 74 65 74 73 6C 65 65 70 20 stacknotetsleep.
```

Let's select *Edit, Patch program, Apply patches to input file...*



Now that we have successfully patched our program, let's re-run it.

Let's seek out *main\_main* again in the function tree.

Function name
 fmt_ptr_fmt_fmtBs
 fmt_ptr_fmt_fmtSbx
 fmt_ptr_fmt_fmtQ
 fmt_ptr_fmt_fmtC
 fmt_ptr_fmt_fmtQc
 fmt_ptr_fmt_fmtFloat
 fmt_ptr_buffer_writeRune
 fmt_glob_func1
 fmt_newPrinter
 <b>fmt_ptr_pp_free</b>
 <b>fmt_ptr_pp_Write</b>
 fmt_Fprintln
 fmt_getField
 fmt_ptr_pp_unknownType
 fmt_ptr_pp_badVerb
 fmt_ptr_pp_fmtBool
 fmt_ptr_pp_fmt0x64
 fmt_ptr_pp_fmtInteger
 <b>fmt_ptr_pp_fmtFloat</b>
 <b>fmt_ptr_pp_fmtComplex</b>
 <b>fmt_ptr_pp_fmtString</b>
 <b>fmt_ptr_pp_fmtBytes</b>
 <b>fmt_ptr_pp_fmtPointer</b>
 <b>fmt_ptr_pp_catchPanic</b>
 <b>fmt_ptr_pp_handleMethods</b>
 <b>fmt_ptr_pp_handleMethods_func4</b>
 <b>fmt_ptr_pp_handleMethods_func3</b>
 <b>fmt_ptr_pp_handleMethods_func2</b>
 <b>fmt_ptr_pp_handleMethods_func1</b>
 <b>fmt_ptr_pp_printArg</b>
 <b>fmt_ptr_pp_printValue</b>
 <b>fmt_ptr_pp_doPrintln</b>
 <b>fmt_init</b>
 <b>type_eq_fmt_fmt</b>
 <b>main_main</b>

Here we can see our revised function.

```

.text:000000000048E480      cmp     rsp, [r14+10h]
.text:000000000048E484      jbe     short loc_48E4DC
.text:000000000048E486      sub     rsp, 40h
.text:000000000048E48A      mov     [rsp+40h+var_8], rbp
.text:000000000048E48F      lea     rbp, [rsp+40h+var_8]
.text:000000000048E494      movups  [rsp+40h+var_18], xmm15
.text:000000000048E49A      lea     rdx, RTYPE_string
.text:000000000048E4A1      mov     qword ptr [rsp+40h+var_18], rdx
.text:000000000048E4A6      lea     rdx, off_4CB850 ; "hacky distributed system world"
.text:000000000048E4AD      mov     qword ptr [rsp+40h+var_18+8], rdx
.text:000000000048E4B2      mov     rbx, cs:os_Stdout
.text:000000000048E4B9      lea     rax, go_itab__os_File_io_Writer
.text:000000000048E4C0      lea     rcx, [rsp+40h+var_18]
.text:000000000048E4C5      mov     edi, 1
.text:000000000048E4CA      mov     rsi, rdi
.text:000000000048E4CD      call    fmt_Fprintln
.text:000000000048E4D2      mov     rbp, [rsp+40h+var_8]
.text:000000000048E4D7      add     rsp, 40h
.text:000000000048E4DB      retn

.text:000000000048E4DC ; -----
.text:000000000048E4DC      loc_48E4DC:                                ; CODE XREF: main_main+4↑j
.text:000000000048E4DC      nop     dword ptr [rax+00h]
.text:000000000048E4E0      call    runtime_morestack_noctxt
.text:000000000048E4E5      jmp     short main_main
.text:000000000048E4E5      main_main      endp

```

Let's set a breakpoint by pressing F2 on the call to *fmt\_Fprintln*.

```

.text:000000000048E480      cmp     rsp, [r14+10h]
.text:000000000048E484      jbe     short loc_48E4DC
.text:000000000048E486      sub     rsp, 40h
.text:000000000048E48A      mov     [rsp+40h+var_8], rbp
.text:000000000048E48F      lea     rbp, [rsp+40h+var_8]
.text:000000000048E494      movups  [rsp+40h+var_18], xmm15
.text:000000000048E49A      lea     rdx, RTYPE_string
.text:000000000048E4A1      mov     qword ptr [rsp+40h+var_18], rdx
.text:000000000048E4A6      lea     rdx, off_4CB850 ; "hacky distributed system world"
.text:000000000048E4AD      mov     qword ptr [rsp+40h+var_18+8], rdx
.text:000000000048E4B2      mov     rbx, cs:os_Stdout
.text:000000000048E4B9      lea     rax, go_itab__os_File_io_Writer
.text:000000000048E4C0      lea     rcx, [rsp+40h+var_18]
.text:000000000048E4C5      mov     edi, 1
.text:000000000048E4CA      mov     rsi, rdi
.text:000000000048E4CD      call    fmt_Fprintln
.text:000000000048E4D2      mov     rbp, [rsp+40h+var_8]
.text:000000000048E4D7      add     rsp, 40h
.text:000000000048E4DB      retn

.text:000000000048E4DC ; -----
.text:000000000048E4DC      loc_48E4DC:                                ; CODE XREF: main_main+4↑j
.text:000000000048E4DC      nop     dword ptr [rax+00h]
.text:000000000048E4E0      call    runtime_morestack_noctxt
.text:000000000048E4E5      jmp     short main_main
.text:000000000048E4E5      main_main      endp

```

Finally let's debug!

We hit our breakpoint.

```
.text:0000000000B7E4C5 mov     edi, 1
.text:0000000000B7E4CA mov     rsi, rdi
.text:0000000000B7E4CD call    fmt_Fprintln
.text:0000000000B7E4D2 mov     rbp, [rsp+40h+var_8]
.text:0000000000B7E4D7 add     rsp, 40h
```

Let's step over the call and watch what happens in the console window.

A screenshot of a Windows console window. The title bar shows the file path 'C:\Users\mytec\Documents\C' and standard window controls. The console output displays the text 'hacky distributed system world' on a single line.

Success!

In our next lesson we will begin to understand primitive types in Go.

## Chapter 4: Primitive Types

Golang has three basic types which are *bool*, *numeric* and *string*.

Once a variable is declared it is automatically populated with a null value.

Let's create a new project and get started by following the below steps.

New File  
main.go

Now let's populate our **main.go** file with the following.

```
package main

import "fmt"

func main() {
    b := true
    i := 42
    f := 3.14
    s := "42"

    fmt.Println("bool: ", b)
    fmt.Println("int: ", i)
    fmt.Println("float32: ", f)
    fmt.Println("string: ", s)
}
```

Let's open up the terminal by click CTRL+SHIFT+` and type the following.

```
go mod init main
go mod tidy
go build
Let's run the binary!
```

```
.\main.exe
```

Output...

```
bool:  true
int:  42
float32:  3.14
string:  42
```

We can clearly see the respective values and how Golang handles them. In our next lesson we will debug this simple program.