Chapter 4

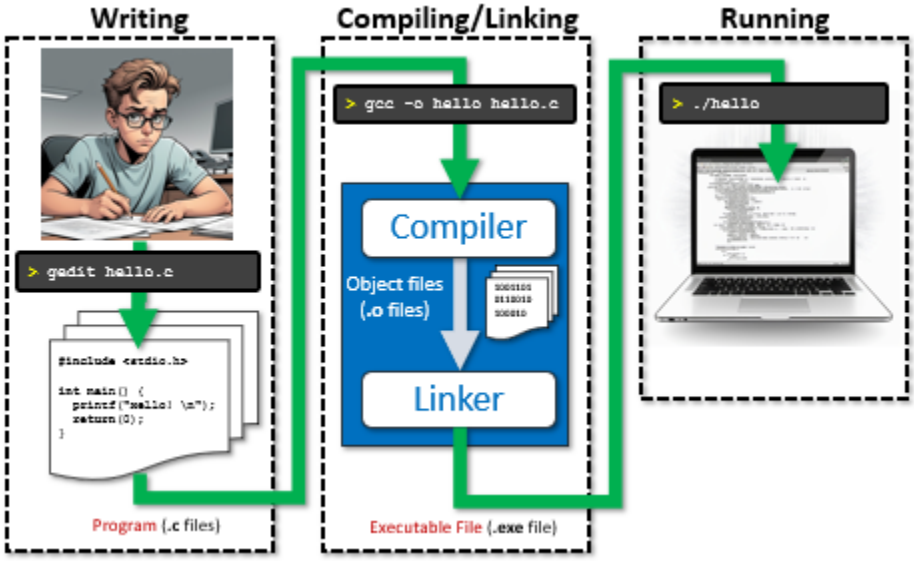
Compiling and Runtime Optimization

What is in This Chapter ?

This chapter explains the details behind what happens in the process of transforming our source code into executable files. It does this by explaining the difference between **compiling**, **assembling** and **linking**. It explains the difference between **assembly code** and **machine code**. It explains how to create simple **makefiles** in order to simplify the process of creating our C programs. Finally, we will have a brief discussion of optimizing our code by understanding registers and coding in assembly language.

4.1 The Compilation Process

Recall the steps involved in getting a program from your head into a running/working executable:



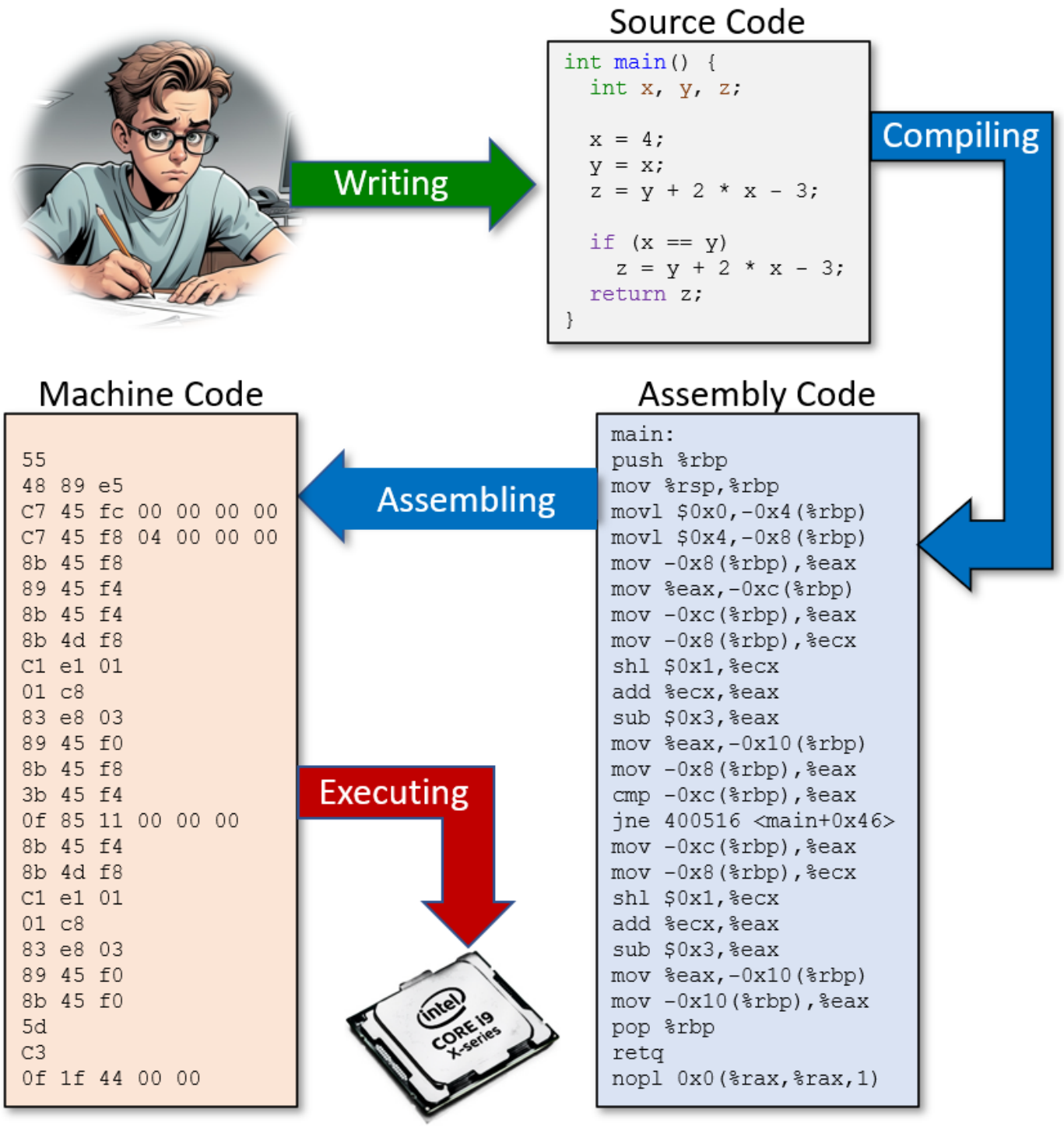
We are going to discuss a little bit more about what happens in the blue box above. That is, we will talk about how the compiler takes your written code and produces an executable file.

A computer's CPU (Central Processing Unit) is the part of your computer that carries out the instructions specified by your program. It does this by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by your instructions. There are many types of CPUs out there, each having their own pros and cons with respect to their capabilities.

Modern CPUs are Microprocessors … in that all the work happens on a single integrated circuit (IC). Most microprocessors these days have multiple cores … that is … multiple processors (i.e., CPUs) running at the same time in order to process the instructions more quickly (i.e., "many hands make light work"). We will not discuss this further here, but instead we will focus on what happens with a single-core processor … that is … a single CPU.

It would be nice and simple if the CPU could take our source code and simply run our program by reading it. However, things are not so simple. Processors have a very limited number of simple/basic commands and operations that they can perform. This list is called its ***instruction set***. You can find the instruction set for a CPU by looking up a CPU's software developer manual. But these can be large (intel has one which is around 2200 pages!!). It is way beyond the scope of this course to get into these details. Likely it would take two or three dedicated courses to learn the instruction set of just one type of processor. All that we are interested in at the moment is understanding that there is quite a lot that happens in order to get your nicely-readable code into what is required by the CPU.

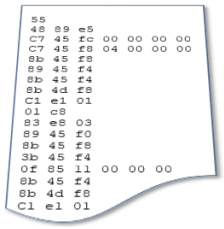
Here is what happens, typically:



,

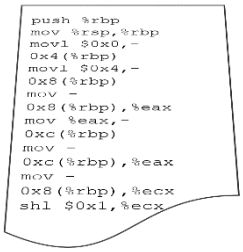
As you can see in the diagram, there really are 3 languages involved in this process. We write our code in **C** and then the compiler *compiles* (i.e., translates) our code into **assembly code** which is a set of many more much simpler commands along with our data. This assembly code is then *assembled* into **machine code** … which is a sequence of numbers corresponding to instructions that the CPU understands, along with data for those instructions. The CPU essentially gets a stream of bytes sent to it and just *executes* (i.e., processes) these numbers one at a time.

Machine Code:



The machine code is very low level and is in a format that the CPU understands. It is really just a set of numbers that represent specific CPU instructions and the accompanying data required for that instruction. The specific instruction codes will be different for each CPU and operating system. Therefore, machine code that was created on one platform (i.e., CPU and OS combination) will NOT run on a different platform. Machine code in C manifests itself as *object files* which we can generate when we use the **-c** option in our **gcc** compiler. It is too difficult for us humans to work (i.e., to write programs) at the low level of machine code, so we let the compiler construction programmers do that and we just make use of their compilers. Keep in mind that some compilers can produce very efficient/fast machine code while others may be slower. It can be somewhat of an artform to be able to take advantage of the CPU with fancy compiling tricks. Maybe you can take a course one day on compiler construction.

Assembly Code:



It is slightly easier to work with assembly code, as the instructions are more readable (e.g., **add**, **sub**, **mov**) but still … things are so low-level that it is difficult to avoid getting bogged down in low-level details. We would rather concentrate on the high-level aspects of what our program is supposed to accomplish. Like machine code, assembly code is platform-specific … so what we compile on one platform will NOT run on another platform.

The Compilation Process – Step 1: PreProcessing

Let us now look a little closer at compiling. The compiling process begins with a ***preprocessing*** stage. That is, it goes through your source code and does a few things that will make its life easier when it goes to produce the machine code.

Your C code typically contains some preprocessing directives … which is any code that begins with the # symbol. We have seen these, for example:

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

#define MAX_STR                  32
#define NUMBER_OF_SCORES          8
#define NUMBER_OF_PERFORMANCES    3
```

There are other directives as well such as **#undef**, **#if**, **#ifdef**, **#ifndef**, **#error**, etc…
For the **#include** directive, the compiler will take the source code from the included header file and insert it right into your file as if you wrote it there yourself. It is simply a text substitution. It helps keep our source code more compact by not having to cut/paste the contents of those header files each time that we use them. For the **#define** directive, the compiler will substitute that constant's value in each place that the defined constant's name appears.

For an example of how the including of a header file works, let us write our own header file. Here is one with a few definitions which is stored in a file called **myDefinitions.h**:

Code from **myDefinitions.h**

```
#define DAYS_OF_WEEK   7
#define PI             3.14159
#define MAX            15
```

Header files can contain constant definitions, data type definitions, function prototypes (a.k.a. signatures) … but it may not contain function implementations. Since this is our own header file, we will place it in the same directory as our code in order to keep things simple. In order to include a header file from the current directory, as opposed to one from a library, we use double quotes instead of <> as follows:

```
#include "myDefinitions.h"
```

Here is a test program that includes this header file. The code behaves as if the definitions were at the top of this C program file:

Code from **headerExample.c**

```
#include <stdio.h>
#include "myDefinitions.h"

int main() {
  int  x = MAX;
  int *y = &x;

  printf("x + *y = %d\n", x + *y);
  printf("Days = %d, PI = %f\n", DAYS_OF_WEEK, PI);
}
```

Output

```
x + *y = 30
Days = 7, PI = 3.141590
```

The preprocessing stage considers each **.c** source file individually, and produces new source code with the text substitutions, along with some other stuff necessary for the compiling process. You can compile with the **-E** option to see the result of preprocessing.

```
student@COMPBase:~$ gcc -E headerExample.c
student@COMPBase:~$ ... lots will be printed …
```

Here, for example, is the preprocessing result of the above code, which will appear at the end of the output when you use the **-E** option:

```
int main() {
  int  x = 15;
  int *y = &x;

  printf("x + *y = %d\n", x + *y);
  printf("Days = %d, PI = %f\n", 7, 3.14159);
}
```

Directives such as **#ifdef** and **#ifndef** allow you to include or eliminate chunks of code during the compilation process without having to comment out your code temporarily. This is good for debugging. Consider this code, where a **DEBUG** flag is enabled or disabled:

Code from **ifdefExample.c**

```
#include <stdio.h>
#include "myDefinitions.h"

#define DEBUG 1

int main() {
  int  x = MAX;
  int *y = &x;

  #ifdef DEBUG
    printf("*y = %d\n", *y);
  #endif
  printf("x + *y = %d\n", x + *y);
  printf("Days = %d, PI = %f\n", DAYS_OF_WEEK, PI);
}
```

Output

```
*y = 15

x + *y = 30
Days = 7, PI = 3.141590
```

```
#include <stdio.h>
#include "myDefinitions"

// #define DEBUG 1

int main() {
  int  x = MAX;
  int *y = &x;

  #ifdef DEBUG
    printf("*y = %d\n", *y);
  #endif
  printf("x + *y = %d\n", x + *y);
  printf("Days = %d, PI = %f\n", DAYS_OF_WEEK, PI);
}
```

Commented out now.

Not printed now.

```
x + *y = 30
Days = 7, PI = 3.141590
```

As you can imagine … if you place the **#ifdef** directive before all your debugging statements, then you will be able to enable/disable all of them by simply commenting in/out that single line of code.

Here is the result of doing **gcc -E ifdefExample.c** with and without the **DEBUG** definition:

With DEBUG defined

```
int main() {
  int  x = 15;
  int *y = &x;

  printf("*y = %d\n", *y);

  printf("x + *y = %d\n", x + *y);
  printf("Days = %d, PI = %f\n", 7, 3.14159);
}
```

Without DEBUG defined

```
int main() {
  int  x = 15;
  int *y = &x;


  printf("x + *y = %d\n", x + *y);
  printf("Days = %d, PI = %f\n", 7, 3.14159);
}
```

The Compilation Process – Step 2: Assembly Code Creation

Once the preprocessing has been done on the source code, the compiler then translates the code into assembly code. In doing so, it attempts to optimize the code as much as it can. It will resolve internal function addresses (i.e., functions for this file). That means, the compiler will, among other things, determine where each function will reside in memory and make sure that instructions are set up to make the function calls at the right time. You can use **gcc -S** (note: the S must be capitalized) to see the resulting assembly code:

```
student@COMPBase:~$ gcc -S ifdefExample.c
student@COMPBase:~$
```

It will produce an assembly code file (in this case called **ifdefExample.s**) like this:

```
.file  "ifdefExample.c"
       .text
       .section      .rodata
.LC0:
       .string       "*y = %d\n"
.LC1:
       .string       "x + *y = %d\n"
.LC3:
       .string       "Days = %d, PI = %f\n"
       .text
       .globl main
       .type  main, @function
main:
.LFB0:
       .cfi_startproc
       pushq  %rbp
       .cfi_def_cfa_offset 16
       .cfi_offset 6, -16
       movq   %rsp, %rbp
       .cfi_def_cfa_register 6
       subq   $48, %rsp
       movq   %fs:40, %rax
       movq   %rax, -8(%rbp)
       xorl   %eax, %eax
       movl   $15, -20(%rbp)
       leaq   -20(%rbp), %rax
       movq   %rax, -16(%rbp)
       movq   -16(%rbp), %rax
       movl   (%rax), %eax
       movl   %eax, %esi
       leaq   .LC0(%rip), %rdi
       movl   $0, %eax
       call   printf@PLT
       movq   -16(%rbp), %rax
       movl   (%rax), %edx
       movl   -20(%rbp), %eax
       addl   %edx, %eax
       movl   %eax, %esi
       leaq   .LC1(%rip), %rdi
       movl   $0, %eax
       call   printf@PLT
       movq   .LC2(%rip), %rax
       movq   %rax, -40(%rbp)
       movsd  -40(%rbp), %xmm0
       movl   $7, %esi
       leaq   .LC3(%rip), %rdi
       movl   $1, %eax
```

```
        call    printf@PLT
        movl    $0, %eax
        movq    -8(%rbp), %rcx
        xorq    %fs:40, %rcx
        je      .L3
        call    __stack_chk_fail@PLT
.L3:
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size  main, .-main
        .section      .rodata
        .align 8
.LC2:
        .long  4028335726
        .long  1074340345
        .ident "GCC: (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0"
        .section      .note.GNU-stack,"",@progbits
```

As you can see … the code has many more commands … but they are simple instructions.

The Compilation Process – Step 3: Machine Code Creation (Assembling)

The final stage in our code transformation is to take the assembly code and assemble it into machine code so that we can send the commands to the CPU directly. This is very tedious work that we would not want to do manually. The basic idea is to map each command to a byte code as well as ensuring that the commands are all evaluated in the correct order. You can use the **gcc -c** option to produce the **.o** object file representing the machine code:

```
student@COMPBase:~$ gcc -c ifdefExample.c
student@COMPBase:~$
```

It will produce a machine code file (in this case called **ifdefExample.o**) which you can view in emacs. To view it … start up emacs then press **ESC** key to enter command mode. Then type **x** to specify hex mode … the command prompt at the bottom will then show M-x. type **hexl-find-file** and press **ENTER** key. Then type in the **ifdefExample.o** filename (for our example).

You should see this below:

```
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0100 3e00 0100 0000 0000 0000 0000 0000  ..>.............
00000020: 0000 0000 0000 0000 d003 0000 0000 0000  ................
00000030: 0000 0000 4000 0000 0000 4000 0e00 0d00  ....@.....@.....
00000040: f30f 1efa 5548 89e5 4883 ec20 6448 8b04  ....UH..H.. dH..
00000050: 2528 0000 0048 8945 f831 c0c7 45ec 0f00  %(...H.E.1..E...
00000060: 0000 488d 45ec 4889 45f0 488b 45f0 8b00  ..H.E.H.E.H.E...
00000070: 89c6 488d 0500 0000 0048 89c7 b800 0000  ..H......H......
00000080: 00e8 0000 0000 488b 45f0 8b10 8b45 ec01  ......H.E....E..
00000090: d089 c648 8d05 0000 0000 4889 c7b8 0000  ...H......H.....
000000a0: 0000 e800 0000 0048 8b05 0000 0000 6648  .......H......fH
000000b0: 0f6e c0be 0700 0000 488d 0500 0000 0048  .n......H......H
000000c0: 89c7 b801 0000 00e8 0000 0000 b800 0000  ................
000000d0: 0048 8b55 f864 482b 1425 2800 0000 7405  .H.U.dH+.%(...t.
000000e0: e800 0000 00c9 c300 2a79 203d 2025 640a  ........*y = %d.
000000f0: 0078 202b 202a 7920 3d20 2564 0a00 4461  .x + *y = %d..Da
00000100: 7973 203d 2025 642c 2050 4920 3d20 2566  ys = %d, PI = %f
00000110: 0a00 0000 0000 0000 6e86 1bf0 f921 0940  ........n....!.@
00000120: 0047 4343 3a20 2855 6275 6e74 7520 3131  .GCC: (Ubuntu 11
00000130: 2e34 2e30 2d31 7562 756e 7475 317e 3232  .4.0-1ubuntu1~22
00000140: 2e30 3429 2031 312e 342e 3000 0000 0000  .04) 11.4.0.....
00000150: 0400 0000 1000 0000 0500 0000 474e 5500  ............GNU.
00000160: 0200 00c0 0400 0000 0300 0000 0000 0000  ................
```

```
00000170: 1400 0000 0000 0000 017a 5200 0178 1001  .........zR..x..
00000180: 1b0c 0708 9001 0000 1c00 0000 1c00 0000  ................
00000190: 0000 0000 a700 0000 0045 0e10 8602 430d  .........E....C.
000001a0: 0602 9e0c 0708 0000 0000 0000 0000 0000  ................
000001b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001c0: 0100 0000 0400 f1ff 0000 0000 0000 0000  ................
000001d0: 0000 0000 0000 0000 0000 0000 0300 0100  ................
000001e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001f0: 0000 0000 0300 0500 0000 0000 0000 0000  ................
00000200: 0000 0000 0000 0000 1000 0000 1200 0100  ................
00000210: 0000 0000 0000 0000 a700 0000 0000 0000  ................
00000220: 1500 0000 1000 0000 0000 0000 0000 0000  ................
00000230: 0000 0000 0000 0000 1c00 0000 1000 0000  ................
00000240: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000250: 0069 6664 6566 4578 616d 706c 652e 6300  .ifdefExample.c.
00000260: 6d61 696e 0070 7269 6e74 6600 5f5f 7374  main.printf.__st
00000270: 6163 6b5f 6368 6b5f 6661 696c 0000 0000  ack_chk_fail....
00000280: 3500 0000 0000 0000 0200 0000 0300 0000  5...............
00000290: fcff ffff ffff ffff 4200 0000 0000 0000  ........B.......
000002a0: 0400 0000 0500 0000 fcff ffff ffff ffff  ................
000002b0: 5600 0000 0000 0000 0200 0000 0300 0000  V...............
000002c0: 0500 0000 0000 0000 6300 0000 0000 0000  ........c.......
000002d0: 0400 0000 0500 0000 fcff ffff ffff ffff  ................
000002e0: 6a00 0000 0000 0000 0200 0000 0300 0000  j...............
000002f0: 2c00 0000 0000 0000 7b00 0000 0000 0000  ,.......{.......
00000300: 0200 0000 0300 0000 1200 0000 0000 0000  ................
00000310: 8800 0000 0000 0000 0400 0000 0500 0000  ................
00000320: fcff ffff ffff ffff a100 0000 0000 0000  ................
00000330: 0400 0000 0600 0000 fcff ffff ffff ffff  ................
00000340: 2000 0000 0000 0000 0200 0000 0200 0000   ...............
00000350: 0000 0000 0000 0000 002e 7379 6d74 6162  ..........symtab
00000360: 002e 7374 7274 6162 002e 7368 7374 7274  ..strtab..shstrt
00000370: 6162 002e 7265 6c61 2e74 6578 7400 2e64  ab..rela.text..d
00000380: 6174 6100 2e62 7373 002e 726f 6461 7461  ata..bss..rodata
00000390: 002e 636f 6d6d 656e 7400 2e6e 6f74 652e  ..comment..note.
000003a0: 474e 552d 7374 6163 6b00 2e6e 6f74 652e  GNU-stack..note.
000003b0: 676e 752e 7072 6f70 6572 7479 002e 7265  gnu.property..re
000003c0: 6c61 2e65 685f 6672 616d 6500 0000 0000  la.eh_frame.....
000003d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000003e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000003f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000400: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000410: 2000 0000 0100 0000 0600 0000 0000 0000   ...............
00000420: 0000 0000 0000 0000 4000 0000 0000 0000  ........@.......
00000430: a700 0000 0000 0000 0000 0000 0000 0000  ................
00000440: 0100 0000 0000 0000 0000 0000 0000 0000  ................
00000450: 1b00 0000 0400 0000 4000 0000 0000 0000  ........@.......
00000460: 0000 0000 0000 0000 8002 0000 0000 0000  ................
00000470: c000 0000 0000 0000 0b00 0000 0100 0000  ................
00000480: 0800 0000 0000 0000 1800 0000 0000 0000  ................
00000490: 2600 0000 0100 0000 0300 0000 0000 0000  &...............
000004a0: 0000 0000 0000 0000 e700 0000 0000 0000  ................
000004b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000004c0: 0100 0000 0000 0000 0000 0000 0000 0000  ................
000004d0: 2c00 0000 0800 0000 0300 0000 0000 0000  ,...............
000004e0: 0000 0000 0000 0000 e700 0000 0000 0000  ................
000004f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000500: 0100 0000 0000 0000 0000 0000 0000 0000  ................
00000510: 3100 0000 0100 0000 0200 0000 0000 0000  1...............
00000520: 0000 0000 0000 0000 e800 0000 0000 0000  ................
00000530: 3800 0000 0000 0000 0000 0000 0000 0000  8...............
00000540: 0800 0000 0000 0000 0000 0000 0000 0000  ................
00000550: 3900 0000 0100 0000 3000 0000 0000 0000  9.......0.......
00000560: 0000 0000 0000 0000 2001 0000 0000 0000  ........ .......
00000570: 2c00 0000 0000 0000 0000 0000 0000 0000  ,...............
00000580: 0100 0000 0000 0000 0100 0000 0000 0000  ................
00000590: 4200 0000 0100 0000 0000 0000 0000 0000  B...............
000005a0: 0000 0000 0000 0000 4c01 0000 0000 0000  ........L.......
000005b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000005c0: 0100 0000 0000 0000 0000 0000 0000 0000  ................
000005d0: 5200 0000 0700 0000 0200 0000 0000 0000  R...............
000005e0: 0000 0000 0000 0000 5001 0000 0000 0000  ........P.......
000005f0: 2000 0000 0000 0000 0000 0000 0000 0000   ...............
00000600: 0800 0000 0000 0000 0000 0000 0000 0000  ................
00000610: 6a00 0000 0100 0000 0200 0000 0000 0000  j...............
00000620: 0000 0000 0000 0000 7001 0000 0000 0000  ........p.......
00000630: 3800 0000 0000 0000 0000 0000 0000 0000  8...............
00000640: 0800 0000 0000 0000 0000 0000 0000 0000  ................
00000650: 6500 0000 0400 0000 4000 0000 0000 0000  e.......@.......
00000660: 0000 0000 0000 0000 4003 0000 0000 0000  ........@.......
00000670: 1800 0000 0000 0000 0b00 0000 0900 0000  ................
00000680: 0800 0000 0000 0000 1800 0000 0000 0000  ................
00000690: 0100 0000 0200 0000 0000 0000 0000 0000  ................
000006a0: 0000 0000 0000 0000 a801 0000 0000 0000  ................
000006b0: a800 0000 0000 0000 0c00 0000 0400 0000  ................
000006c0: 0800 0000 0000 0000 1800 0000 0000 0000  ................
```

```
000006d0: 0900 0000 0300 0000 0000 0000 0000 0000  ................
000006e0: 0000 0000 0000 0000 5002 0000 0000 0000  ........P.......
000006f0: 2d00 0000 0000 0000 0000 0000 0000 0000  -...............
00000700: 0100 0000 0000 0000 0000 0000 0000 0000  ................
00000710: 1100 0000 0300 0000 0000 0000 0000 0000  ................
00000720: 0000 0000 0000 0000 5803 0000 0000 0000  ........X.......
00000730: 7400 0000 0000 0000 0000 0000 0000 0000  t...............
00000740: 0100 0000 0000 0000 0000 0000 0000 0000  ................
```

The left side shows the byte numbers in hex (16 bytes per line). The right side shows the ASCII version of the machine code. The center 8 columns shows the machine code in order. Of course, you don't need to know any of this … but it is nice to be able to know how to view machine code. If you understood it, you could actually edit this code and make changes to how the program will run.

The above output is the result from our single object file. The final executable file will be larger, because we usually ***link*** together more than one object file to produce the executable file. This final part of the whole ordeal is called ***linking***. The linking stage resolves external function addresses (i.e., ensures that functions from one file are able to call ones from another file). It also allows us to call the various library functions that we are trying to make use of in our code. So, linking ties everything all together into the final executable file.

Why is it good to separate the compiling and linking stages? Well, it is often the case that our program makes use of many of our own files as well as many pre-compiled library files. By keeping the compilation step separate, when we want to prepare an executable for a program that uses many files, we only need to re-compile code that we just modified. We don't have to re-compile everything from scratch again. It is important not to re-compile unnecessary files, as this wastes much time in the software development process.

We have already been using pre-compiled library functions in our code. We often **#include** a header file so that we can make use of some of the functions defined in an existing library. In addition to including the header file, we also need to "link-in" the library file. The standard C library is in a file called **libc.a** which is always linked-in by default.

There are two types of linking:

1. **Static Linking -** Library code is copied into executable
   - ✗ Increases the size of the executable
   - ✓ Faster execution time

2. **Dynamic Linking** (default setting) - Library code is loaded at runtime
   - ✓ Smaller executable
   - ✗ Slower execution time

Let us look at an example of linking some files together. Consider this header file:

Code from **linkExampleTypes.h**

```
#define MAX_STR 32

typedef struct {
  char first[MAX_STR];
  char last[MAX_STR];
} NameType;

void enterName(NameType *name);
void capFix(char *str);
```

Now consider the main application file and a file with utility (i.e., helper) functions:

Code from **linkExampleMain.c**

```
#include <stdio.h>
#include <string.h>
#include "linkExampleTypes.h"

int main() {
  NameType newName;

  while(1) {
    enterName(&newName);

    if (strcmp(newName.first,"-1") == 0 && strcmp(newName.last,"-1") == 0)
      break;

    capFix(newName.first);
    capFix(newName.last);

    printf("my name is %s %s\n",
           newName.first, newName.last);
  }
}
```

Code from **linkExampleUtil.c**

```
#include <stdio.h>
#include <string.h>
#include "linkExampleTypes.h"

void enterName(NameType *name) {
  printf("\n");
  printf("Enter a name: ");
  scanf("%s %s", name->first, name->last);
}

void capFix(char *str) {
  if (str[0] >= 'a' && str[0] <= 'z')
    str[0] = str[0] - 'a' + 'A';

  for (int i=1; i<strlen(str); i++)
    if (str[i] >= 'A' && str[i] <= 'Z')
      str[i] = str[i] - 'A' + 'a';
}
```

One way to compile this program into an executable is to compile both files individually:

```
student@COMPBase:~$ gcc -c linkExampleMain.c
student@COMPBase:~$ gcc -c linkExampleUtil.c
student@COMPBase:~$ ls
headerExample     ifdefExample.c     linkExampleTypes.h  myDefinitions.h
headerExample.c   linkExampleMain.c  linkExampleUtil.c
ifdefExample      linkExampleMain.o  linkExampleUtil.o
student@COMPBase:~$
```

We can then link the two together using **gcc -o** as follows:

```
student@COMPBase:~$ gcc -o linkExample linkExampleMain.o linkExampleUtil.o
student@COMPBase:~$ ls
headerExample     ifdefExample.c     linkExampleMain.o   linkExampleUtil.o
headerExample.c   linkExample        linkExampleTypes.h  myDefinitions.h
ifdefExample      linkExampleMain.c  linkExampleUtil.c
student@COMPBase:~$
```

Notice that when we use **gcc -o**, we first specify the name of the executable file that we are trying to create … in this case … **linkExample**. Then we supply the compiled object files that we want to link together. As a result, we end up with the runnable executable.

Of course, we can shorten the whole process by using **gcc -o** along with the **.c** source files as we have been doing:

```
student@COMPBase:~$ gcc -o linkExample linkExampleMain.c linkExampleUtil.c
student@COMPBase:~$ ls
headerExample     ifdefExample.c     linkExampleTypes.h
headerExample.c   linkExample        linkExampleUtil.c
ifdefExample      linkExampleMain.c  myDefinitions.h
student@COMPBase:~$
```

In this case, the object files are created and then removed during the process so they do not clutter up the directory.

4.2 Makefiles

When creating large programs that make use of a lot of source files, it can be cumbersome to compile your programs one file at a time. It can also be tedious having to write out each source file name on the script command line when compiling. There is a tool for organizing the compiling/linking commands during the compilation process.

> A ***Makefile*** is a text file that is used by the **make** command to automatically build executable programs and libraries from source code.

When working on larger pieces of software, it is crucial that you understand how to create and use makefiles. There are two main advantages of using makefiles:

- It simplifies the compiling process down to one command
- It keeps track of what needs to be compiled

The makefile manages dependencies between source and header files so that it only recompiles source files that have changed since the last "make". It does this by comparing the timestamp on the source files with the timestamp on the object files. If the source file is newer, it gets recompiled.

Let us look at an example of a basic makefile. Here is a makefile that would be used to compile our **linkExample** code:

Code from **makefile**

```
all:  linkExampleMain.o linkExampleUtil.o
      gcc -o linkExample linkExampleMain.o linkExampleUtil.o

linkExampleMain.o: linkExampleMain.c linkExampleTypes.h
      gcc -c linkExampleMain.c

linkExampleUtil.o: linkExampleUtil.c linkExampleTypes.h
      gcc -c linkExampleUtil.c
```

The makefile is saved in the same directory and is simply called **makefile** (all lowercase letters with no file extension).

Notice that there are three parts to the file, in this case. The left side (in blue) represents labels that refer to the linking stage (i.e., **all**) and the two object file compilation stages (i.e., **linkExampleMain.o** & **linkExampleUtil.o**). They are arbitrary labels, but in order to have the proper dependencies they should be the same name as the object file that will be created by compiling that line.

The first line represents the dependencies for the executable. That is, by means of the two labels **linkExampleMain.o** and **linkExampleUtil.o**, it specifies the dependencies for the executable to be built … that is … the compiling that must take place before the linking. The second line then specifies the **gcc** command that is used to link things together. This is the same command that we used to compile and link previously.

As a side point, to the left of the **gcc** command there MUST BE A TAB character … not spaces. Otherwise, you will get an error something like this:

```
makefile:2: *** missing separator.  Stop.
```

The next line (i.e., 3rd line) represents the files that the main object file depends on to compile. In this case, it is the source file and the header file. Although the **linkExampleUtil.c** file contains functions that are called from within the **linkExampleMain.c** file, we do not need to include the **linkExampleUtil.c** file as a dependency here because the stuff inside those functions does not affect how our **main()** function compiles.

Provided we get the function's signature correct, then all will be ok. That is why we include the function signatures in the header file. The fourth line is the **gcc** command for compiling the **linkExampleMain.c** source file into the **linkExampleMain.o** object file.

The final two lines are the same as the middle two … except that it specifies the dependencies and command for compiling the **lineExampleUtil.o** object file instead of the **lineExampleMain.o** object file.

To "make" the file, we simply use the **make** command as follows:

```
student@COMPBase:~$ make
gcc -c linkExampleMain.c
gcc -c linkExampleUtil.c
gcc -o linkExample linkExampleMain.o linkExampleUtil.o
student@COMPBase:~$ ls
headerExample     ifdefExample.c     linkExampleMain.o    linkExampleUtil.o
headerExample.c   linkExample        linkExampleTypes.h   makefile
ifdefExample      linkExampleMain.c  linkExampleUtil.c    myDefinitions.h
student@COMPBase:~$
```

If you run **make** a second time, you will notice that it does not recompile the object files, but it just links them together:

```
student@COMPBase:~$ make
gcc -o linkExample linkExampleMain.o linkExampleUtil.o
student@COMPBase:~$
```

Makefiles can get quite large and complicated. Sometimes it is nice to simplify the readability of the file. One way to do this is to introduce some variables in the makefile. Consider this variable **OBJ** which allows us to substitute the object file names with the variable name (specified with a $ symbol):

Code from **makefile**

```
OBJ = linkExampleMain.o linkExampleUtil.o

all:  linkExampleMain.o linkExampleUtil.o
      gcc -o linkExample $(OBJ)

linkExampleMain.o: linkExampleMain.c linkExampleTypes.h
      gcc -c linkExampleMain.c

linkExampleUtil.o: linkExampleUtil.c linkExampleTypes.h
      gcc -c linkExampleUtil.c
```

Notice that we defined the variable on the first line to be the string of characters as follows:

```
linkExampleMain.o linkExampleUtil.o
```

Then, whenever we use the variable like this **$(OBJ)**, the string of characters is inserted in its place.

As a result, the makefile can be simpler, although the variable is just used once in our example.

Interestingly, with the labels in the makefile (i.e., the blue ones), we can actually specify one(s) in particular that we want to evaluate. For example, suppose that we want to compile just the **linkExampleMain.c** file. We could just specify the **linkExampleMain.o** label in the make command call like this:

```
student@COMPBase:~$ make linkExampleMain.o
make: 'linkExampleMain.o' is up to date.
student@COMPBase:~$
```

Notice that it tells us that the file is "up to date". That means that it did not re-compile the file because the source code has not been altered since the last time it was compiled. We could then make changes to the **linkExampleMain.c** file and then save it. Then when we try the **make** command again, it will re-compile it:

```
student@COMPBase:~$ make linkExampleMain.o
gcc -c linkExampleMain.c
student@COMPBase:~$
```

If we make changes to the two source files and then just want to compile the **linkExampleMain.c** and **linkExampleUtil.c** files without linking, we can specify both makefile labels like this:

```
student@COMPBase:~$ make linkExampleMain.o linkExampleUtil.o
gcc -c linkExampleMain.c
gcc -c linkExampleUtil.c
student@COMPBase:~$
```

The executable will not be created in these cases.

One more interesting thing that we can do is to add the ability to clean up all the object files and the executable file … leaving just the source files. This is nice to clean up a directory. To do this, we add a **clean** label and use the remove file command **rm** as follows:

Code from **makefile**

```
OBJ = linkExampleMain.o linkExampleUtil.o

all:  linkExampleMain.o linkExampleUtil.o
      gcc -o linkExample $(OBJ)

linkExampleMain.o: linkExampleMain.c linkExampleTypes.h
      gcc -c linkExampleMain.c

linkExampleUtil.o: linkExampleUtil.c linkExampleTypes.h
      gcc -c linkExampleUtil.c

clean:
      rm -f $(OBJ) linkExample
```

This is evaluated when doing a **make clean**. (**-f** means "force" in cases where the file may or may not exist. It also allows ignoring of permissions)

Then to clean up the files we just do **`make clean`**:

```
student@COMPBase:~$ make
gcc -c linkExampleMain.c
gcc -c linkExampleUtil.c
gcc -o linkExample linkExampleMain.o linkExampleUtil.o
student@COMPBase:~$ ls
headerExample     ifdefExample.c      linkExampleMain.o   linkExampleUtil.o
headerExample.c   linkExample         linkExampleTypes.h  makefile
ifdefExample      linkExampleMain.c   linkExampleUtil.c   myDefinitions.h

student@COMPBase:~$ make clean
rm -f linkExampleMain.o linkExampleUtil.o linkExample

student@COMPBase:~$ ls
headerExample     ifdefExample.c       linkExampleUtil.c
headerExample.c   linkExampleMain.c    makefile
ifdefExample      linkExampleTypes.h   myDefinitions.h
student@COMPBase:~$
```

Object files and executable are now gone.

Makefiles can get quite large and even complicated. Here is an old makefile of mine with a few things that I would like to point out:

An older example makefile

```
#########################################################################
# This is the makefile for the Shortest Path Approximation Program      #
#########################################################################


#####################################
# Here are the different C compilers #
#####################################
HOST_CC = gcc -g
i860_CC = pgcc.m64b2 -g
t805_CC = tcc


#########################################################
# Here are the necessary Motif libraries and INCLUDES #
#########################################################
MOTIF_LIB  = -lXm -lXt -lgen -lX11 -lXext
XGL_LIB    = -lxgl
MOTIF_INC  = -I/usr/dt/include
XLIB_INC   = -I/usr/openwin/include
XGL_INC    = -I/opt/SUNWits/Graphics-sw/xgl-3.0/include


#########################################################
# Here are the necessary trollius libraries and INCLUDES #
#########################################################
TROLLIUS_LIB = -L$(TROLLIUSHOME)/lib
TROLLIUS_INC = -I$(TROLLIUSHOME)/h
```

Comments can be made in a makefile by using the #

If the code is to be ported to different platforms, it is nice to have a few compiler options. In the rest of the makefile, we would just have to make the substitution.

It is nice to make variables for various libraries.

```
#######################################################
# Here I list all executables that are to be compiled #
#######################################################
all:    spmain

spmain:       spmain.sun.o tin.sun.o easyMotif.sun.o \
       graph.sun.o spheap.sun.o sp.sun.o sleeve.sun.o funnel.sun.o \
       schemes.sun.o chenhan.sun.o spgui.sun.o
       $(HOST_CC) $(TROLLIUS_INC) $(MOTIF_INC) $(XLIB_INC)\
       -o spmain.sun.o tin.sun.o easyMotif.sun.o \
       graph.sun.o spheap.sun.o sp.sun.o sleeve.sun.o funnel.sun.o \
       schemes.sun.o chenhan.sun.o spgui.sun.o \
       $(TROLLIUS_LIB) $(MOTIF_LIB) $(XGL_LIB) -lt -lm -L../../vdm -lvdmh


################################################
# This makes the object files #
################################################
spmain.sun.o:   spmain.c easyMotif.h colors.h
                $(HOST_CC) $(TROLLIUS_INC) $(MOTIF_INC) $(XLIB_INC) $(XGL_INC)\
                -c spmain.c -o spmain.sun.o

tin.sun.o:      tin.c tin.h
                $(HOST_CC) -c tin.c -o tin.sun.o

easyMotif.sun.o:  easyMotif.c easyMotif.h colors.h
                $(HOST_CC) $(MOTIF_INC) $(XLIB_INC)\
                -c easyMotif.c -o easyMotif.sun.o

graph.sun.o:      graph.c graph.h
                $(HOST_CC) -c graph.c -o graph.sun.o

spheap.sun.o:     spheap.c sp.h
                $(HOST_CC) -c spheap.c -o spheap.sun.o

sleeve.sun.o:     sleeve.c tin.c tin.h sp.h
                $(HOST_CC) -c sleeve.c -o sleeve.sun.o

funnel.sun.o:     funnel.c sleeve.c tin.c tin.h sp.h  vdmlib.h
                $(HOST_CC) $(MOTIF_INC) $(XLIB_INC) $(XGL_INC)\
                -c funnel.c -o funnel.sun.o

spgui.sun.o:      spgui.c sp.h easyMotif.h colors.h
                $(HOST_CC) $(MOTIF_INC) $(XLIB_INC) $(XGL_INC)\
                 -c spgui.c -o spgui.sun.o

schemes.sun.o:    schemes.c sp.h graph.h tin.h
                $(HOST_CC) $(MOTIF_INC) $(XLIB_INC) $(XGL_INC)\
                 -c schemes.c -o schemes.sun.o

sp.sun.o:         sp.c sp.h easyMotif.h colors.h graph.h tin.h
                $(HOST_CC) $(MOTIF_INC) $(XLIB_INC) $(XGL_INC)\
                 -c sp.c -o sp.sun.o

chenhan.sun.o:    chenhan.c funnel.c sleeve.c tin.c chenhan.h tin.h sp.h
                $(HOST_CC) $(MOTIF_INC) $(XLIB_INC) $(XGL_INC)\
                -c chenhan.c -o chenhan.sun.o

#####################################################
# This cleans up all object files and exectuables #
#####################################################
clean:
       rm -f *.o *~ sp
```

Don't forget to include the standard libraries if you need them.

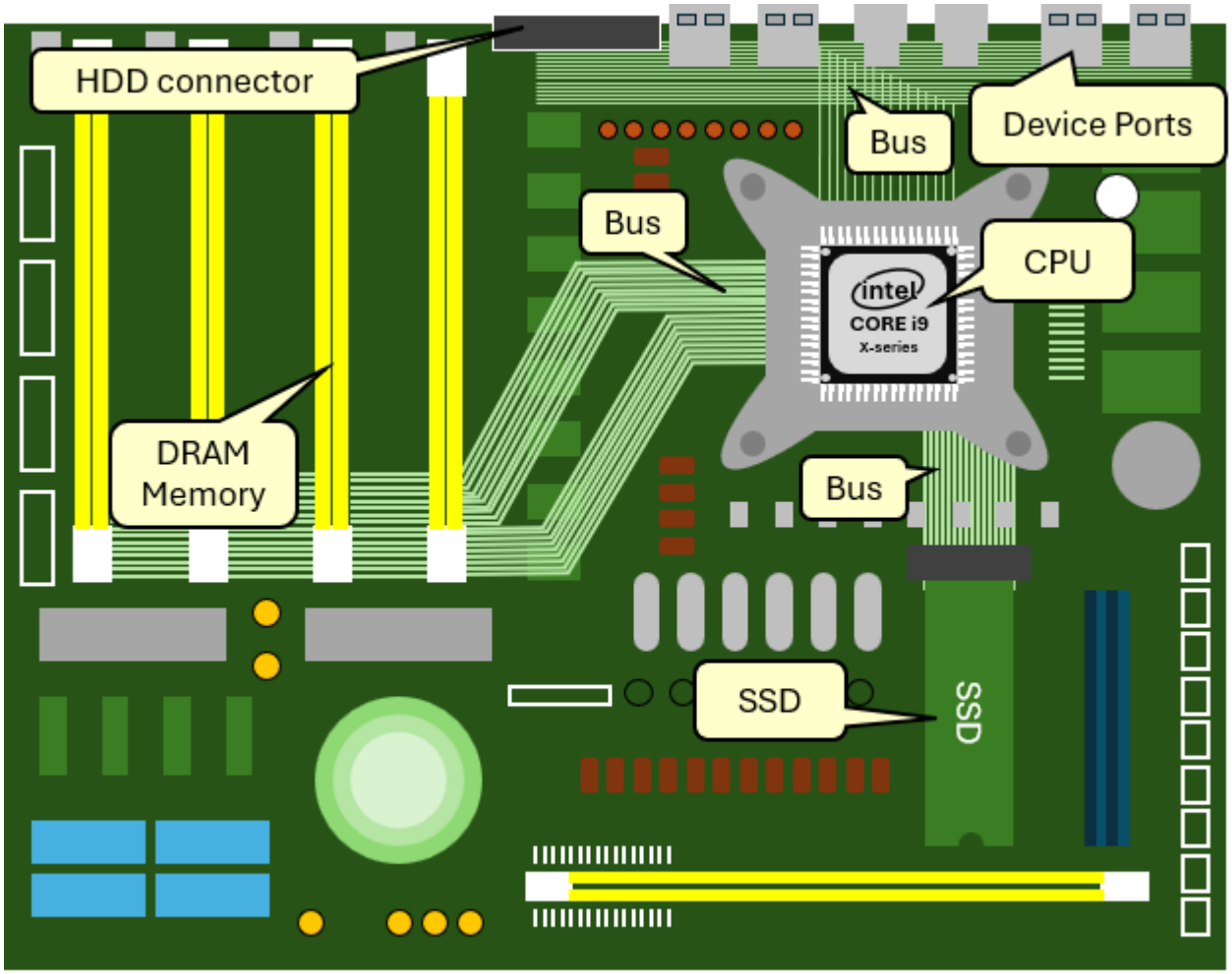
It is good to include ***~** here, which will remove the temporary emacs editor files that were made.

4.3 Runtime Optimization

Most of the time, when coding simple applications in an academic setting, we are not concerned about our program's runtime. However, the speed of our programs in the industrial setting is often of concern because companies can be quite competitive in their products. So, when working for a company, you will want to always produce the most efficient, smooth-running, reliable software possible.

To create fast-running software, we usually have to spend time speeding up our code. Of course, the first step in coding is to produce software that works (e.g., a prototype) as proof-of-concept. But then over time the goal is to fine-tune and speed up the software, making it more efficient. The first step towards speeding up software is to understand the computer system architecture that it is running on.

Here is a depiction of a PC computer motherboard (not based on a real one), showing some components that will be important in our discussion of optimization:



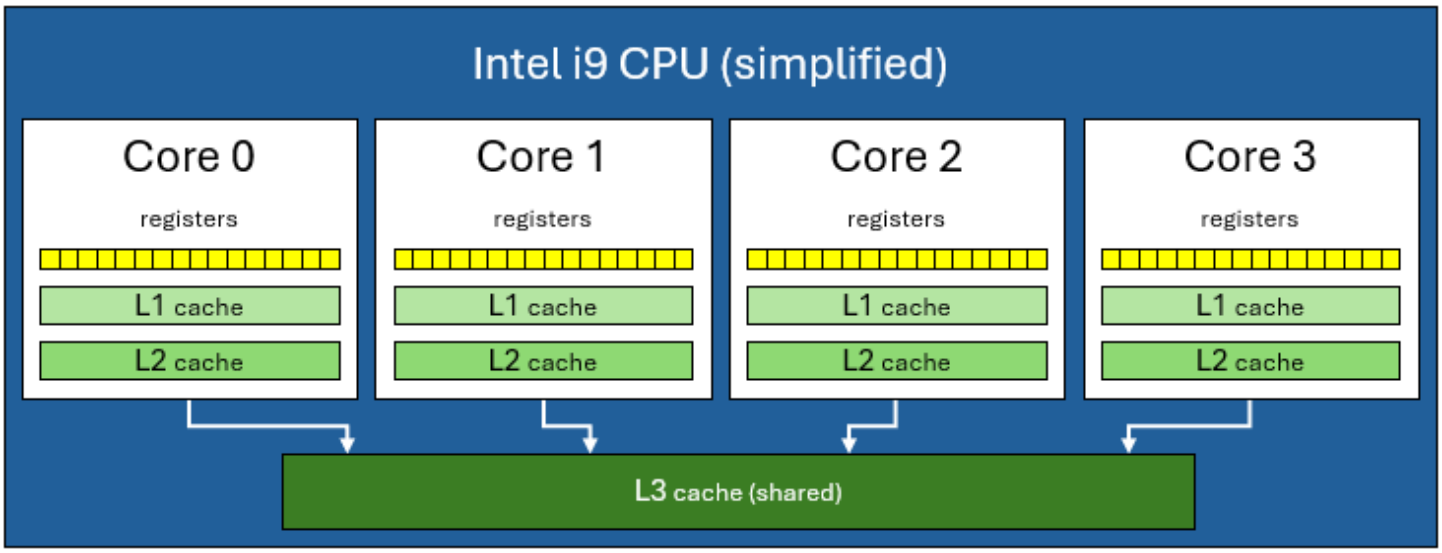
Notice that the **CPU** uses computer ***buses*** to connect to (1) the computer's **DRAM** (dynamic random access) memory, (2) **SSD** (solid state drives), and (3) various device ports, including **HDD** (Hard Disk Drives). Take note that the **DRAM** memory and the **SSD** are on the motherboard, while the **HDD** is off the motherboard (although you can have external **SSD** drives off the motherboard as well).

Why be concerned about what is on/off the motherboard? Well, as it turns out, the **CPU** can do things faster if the data that it needs is close by. This is logical. Imagine that you were sitting at your desk and someone called you on your cellphone and asked you how many books you own. If all the books you own are in front of you on your desk, you could count them quick and give an answer. However, if your books were in the room on a shelf behind you, it would take a few seconds longer to give the answer. If the books were all in another room of your home, it would take even longer because you would have to go there to count them before you give the answer. It could even take a half hour to give the answer if, for example, your books were all at your office at work. And if all your books were in another city altogether, it could take hours. The point is … when data is close by, computations are faster.

As you can probably guess … accessing data stored in **DRAM** is going to be faster than accessing data on a hard drive that is external to the motherboard. Even **SSD**s stored on the motherboard will be slower to access than **DRAM** because of the way that it communicates with the **CPU**. The difference can be orders of magnitude, as we will see soon.

Even with regard to a computer's memory, there can be speed differences depending on the closeness to the **CPU**. Companies that design microprocessors have realized this decades ago and have therefore designed **CPU**s with on-board memory. That is … they have placed electronic memory right on the **CPU** chip itself. This memory takes the form of ***registers*** and ***cache*** memory.

At the time these notes were written, our virtual machine was emulating an **Intel i9** multi-core processor. Therefore, we will discuss the basic architecture of this **CPU** (which itself has many variations). The concepts, however, are similar/common to most computer architectures being used today. Here is a simplified diagram that shows the internal memory of a multi-core **CPU**. This is where we begin to understand what needs to be done to speed up our software.



First, notice that there are multiple ***cores*** to the **CPU**. It is common to have 4 or 8 cores on a **CPU**. Each core is a processing unit that can perform calculations at the same time. Each core has their own ***registers***, ***L1 cache*** and ***L2 cache***. Then there is an ***L3 cache*** that is shared among the cores. All of these are computer memory for storage purposes.

Why so many memory options? They allow access to data at different speeds! The following table gives you a rough idea as to the relative speed differences for each type of memory available:

FASTER

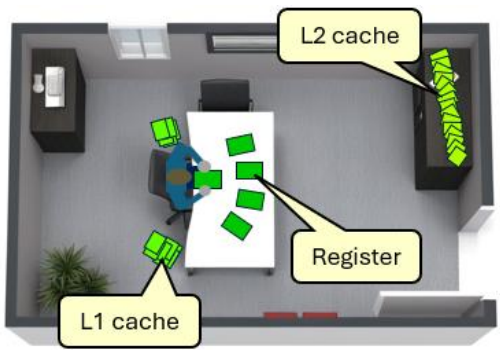
Memory Type	Capacity (bytes)	Speed Scale (cycles)	On CPU?
Registers	256	1	yes
L1 cache	320K	3	yes
L2 cache	2M	12	yes
L3 cache	64M	150	yes
DRAM	8G - 64G	300	no
SSD (solid state drive)	126G - 30T	2,100	no
HDD (hard disk drive)	500G - 4T	7,350	no

SLOWER

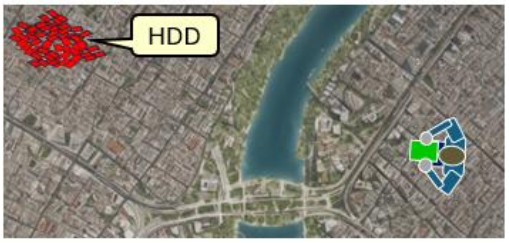
Notice that registers are the fastest to use but the capacity is very small. The **L1 cache** is bigger, but is roughly three times slower than a register. The **L2 cache** is much bigger, but is around 4 times slower than **L1 cache**. The shared **L3 cache** is **50** times slower than the **L1 cache** but has a capacity that is **200x** larger (and it must be shared between cores, so that only one can access it at a time).

As we venture off the **CPU**, we see that **DRAM** is roughly **300x** slower to access than a **register** but has up to **256,000,000x** the capacity!! Finally, notice the hard drives. **SSD** is roughly **2,000x** slower to access than a **register** and **HHD** over **7,000x** slower. Yet their capacity can be arbitrarily large (depending on technology limitations).

To understand the speed differences … let's think about our book-counting problem again. Accessing a **register** is like accessing a book on our desk. **L1 cache** is equivalent to a book on the floor beside us and **L2 cache** is like a book across the room on a shelf. The **L3 cache** is the same as going to another room to access a book … where sometimes the room is being used … so we have to wait:



How does access to **DRAM**, **SSD** and **HDD** compare? Well, accessing **DRAM** is like going to your next-door neighbour's house to access a book. Accessing **SSD** would be the equivalent to going across town to access a book. **HDD** access would be like going to another city to access a book!



Keep in mind that we are discussing a single book access. That means if we want to access 10 **HDD** bytes individually, we would be making 10 city-to-city trips!

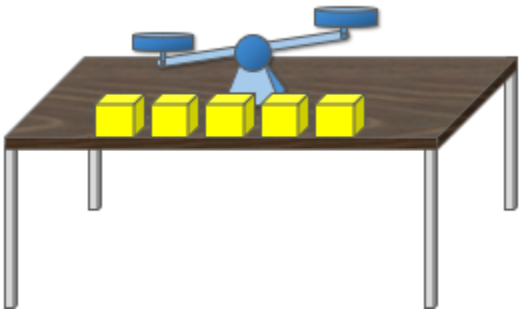
Of course, the speed factors mentioned here are just rough estimates. In reality… a lot goes on behind the scenes when accessing memory and there are many factors that affect performance such as the particular structure of the motherboard, the type of memory cards installed, the speed rating of the hard drives etc..

Let's now discuss the practical things that we can do with our software to speed things up. The trick to efficiency is to keep the *data that we use the most* in a nearby location. The cache memory, for example, is used automatically by the **CPU** to store data that is accessed more often so that access time is kept lower, resulting in improved speed performance. However, when writing our application and systems software, we do not get to specify what data should be kept in the cache. As it turns out, we can really only specify which data is to be stored in registers, **DRAM** and hard disks. Assuming that we are not storing or accessing files, we really only get to decide what should be stored in the registers.

Let's now consider a simply program that we can use to compare runtime performance between three different computer languages: (1) Java, (2) C and (3) Assembly.

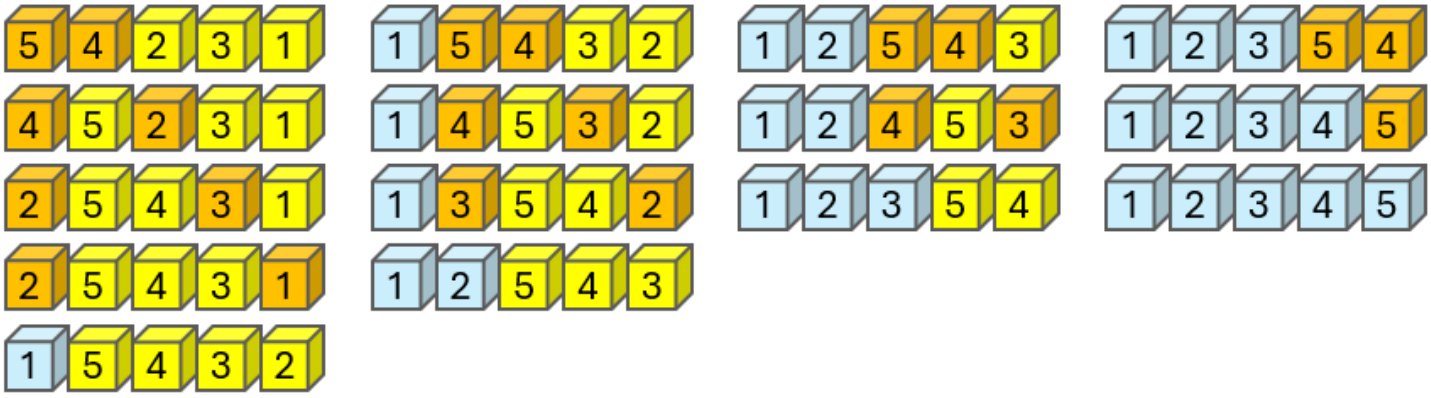
To test speed performance in terms of memory access, we need to have a program that accesses a lot of data repeatedly so that we can see how it all adds up over time. When dealing with assembly code, we will need to understand how things break down into simple steps. Therefore, we will consider a real-world example of sorting a series of boxes in increasing order of weight.

Imagine that you have 5 same-size boxes on a table and each box has a different weight in it. We are also given a balance scale to compare the weight of any two boxes. Now imagine that we can only use one hand to solve this problem. How do we go about doing it? A simple algorithm would be to find the lightest box and put it aside. Then find the next lightest box … and so on. This is the programming equivalent to repeatedly finding the minimum item in an array.



Since we only have one hand, we would perhaps take the first box and place it on the left side of the scale. Then we would take the second box and place it on the right side of the scale. We see which one weighs the least. Normally, we would then put the heavier box back on the table and then take the next box and put it onto the scale. However, we are trying to create an algorithm that does a lot of memory accessing. So, after we compare any two boxes, we will put them both back on the table

again, making sure that the lightest one is on the left. Then we will start again, taking the leftmost box and the next box over in the lineup. Once we tried all the boxes, we are sure that we found the lightest one, which will be leftmost on the table. and then put the boxes back on the table ensuring that the lightest one is on the left. Then, start the whole process over again, ignoring the leftmost box, since it has been determined to be the lightest. Here is a run of the algorithm with lightest boxes having the smaller number:



The left column shows the first round. The orange boxes are compared and a swap is made since the right box is lighter than the left one. Then the selection of the right box continues to the right, comparing and swapping repeatedly until the lightest box is on the left (shown blue). That box is in the correct location and the process starts again in the 2nd column above with the two orange boxes.

Let's write a program that sorts with a similar algorithm as above but for sorting characters instead of weights. We will create an array of the characters in the English alphabet in reverse order:

```
char box[26] = {"ZYXWVUTSRQPONMLKJIHGFEDCBA"};
```

We will then use our sort algorithm on the array so that the letters end up in alphabetic order. The code is easy to write in C as follows:

```
int main() {
   char boxes[26] = {"ZYXWVUTSRQPONMLKJIHGFEDCBA"};

   for (int lightestBoxNum=0; lightestBoxNum<25; lightestBoxNum++) {
      for (int compareBox= lightestBoxNum+1; compareBox<26; compareBox++) {
         char leftWeight = boxes[lightestBoxNum];
         char rightWeight = boxes[compareBox];

         if (rightWeight < leftWeight) {
            boxes[lightestBoxNum] = rightWeight;
            boxes[compareBox] = leftWeight;
         }
      }
   }
   for (int i=0; i<26; i++)
      printf("%c", boxes[i]);
   printf("\n");
}
```

The code will reverse the characters of the array and then display the array. However, the code finishes too quickly, so this will not be useful for comparing with java and assembly. We need to do many more memory accesses so that we can get meaningful timing results. We will adjust the code so that we repeat this sorting many times. Since the characters will be sorted after the first round, we will alternate between sorting them in *alphabetic* and *reverse alphabetic* order so that we always have something to sort. Note the additional highlighted lines of code:

Code from **cSort.c**

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

int main() {
   char boxes[26] = {"ZYXWVUTSRQPONMLKJIHGFEDCBA"};

   for (int repeat=5; repeat>0; repeat--) {
      for (int lightestBoxNum=0; lightestBoxNum<25; lightestBoxNum++) {
         for (int compareBox= lightestBoxNum+1; compareBox<26; compareBox++) {
            char leftWeight = boxes[lightestBoxNum];
            char rightWeight = boxes[compareBox];

            if ((repeat & 1) == 0) {
               if (rightWeight < leftWeight) {
                  boxes[lightestBoxNum] = rightWeight;
                  boxes[compareBox] = leftWeight;
               }
            }
            else {
               if (rightWeight > leftWeight) {
                  boxes[lightestBoxNum] = rightWeight;
                  boxes[compareBox] = leftWeight;
               }
            }
         }
      }
      for (int i=0; i<26; i++)
         printf("%c", boxes[i]);
      printf("\n");
   }
}
```

Output

```
ZYXWVUTSRQPONMLKJIHGFEDCBA
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ZYXWVUTSRQPONMLKJIHGFEDCBA
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ZYXWVUTSRQPONMLKJIHGFEDCBA
```

From the output, we can confirm that the code is continually alternating its sort between increasing and decreasing order. For the final version of the code, we will change the starting repeat value from

5 to **20,000,000** and we will comment out the printf loop at the end so that the program does not produce any output. If we re-compile and run it, it should take about **17** seconds or so to run.

```
student:~$  gcc -o cSort cSort.c
student:~$  ./cSort
student:~$
```

Now let's try a JAVA version of the same code:

Code from **JSort.java**

```
public class JSort {
   public static void main(String args[]) {
      char boxes[] = {'Z','Y','X','W','V','U','T','S','R','Q','P','O','N',
                      'M','L','K','J','I','H','G','F','E','D','C','B','A'};

      for (int repeat=20000000; repeat>0; repeat--) {
         for (int lightestBoxNum=0; lightestBoxNum<25; lightestBoxNum++) {
            for (int compareBox= lightestBoxNum+1; compareBox<26; compareBox++) {
               char leftWeight = boxes[lightestBoxNum];
               char rightWeight = boxes[compareBox];

               if ((repeat & 1) == 0) {
                  if (rightWeight < leftWeight) {
                     boxes[lightestBoxNum] = rightWeight;
                     boxes[compareBox] = leftWeight;
                  }
               }
               else {
                  if (rightWeight > leftWeight) {
                     boxes[lightestBoxNum] = rightWeight;
                     boxes[compareBox] = leftWeight;
                  }
               }
            }
         }
      }
   }
}
```

We compile and run it as follows:

```
student:~$  javac JSort.java
student:~$  java JSort
student:~$
```

The code will only take about **5.5** seconds or so to run!! Why is JAVA running three times faster than our C code? This seems unintuitive since java is an interpreted language … which means that the compiled bytecodes still need to run through the java virtual machine to produce the machine code.

As it turns out … JAVA does a lot of optimizing during the compiling stage. Let's see now how we can speed up our C code in a very simple way.

We can use the **register** keyword which allows us to suggest that the compiler store the variable in a register:

```
register float  counter;
```

Since registers are the closest to the **CPU**, when we perform operations using data within the register it should be much faster than accessing the variable from the **DRAM** memory. A small note though … some compilers may actually ignore our request to store the value in a register 🙁. Here is our adjusted code:

Code from **rSort.c**

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

int main() {
   char boxes[26] = {"ZYXWVUTSRQPONMLKJIHGFEDCBA"};
   register int  repeat;
   register int  lightestBoxNum;
   register int  compareBox;
   register char leftWeight;
   register char rightWeight;

   for (repeat=20000000; repeat>0; repeat--) {
      for (lightestBoxNum =0; lightestBoxNum <25; lightestBoxNum++) {
         for (compareBox= lightestBoxNum+1; compareBox<26; compareBox++) {
            leftWeight = boxes[lightestBoxNum];
            rightWeight = boxes[compareBox];

            if ((repeat & 1) == 0) {
               if (rightWeight < leftWeight) {
                  boxes[lightestBoxNum] = rightWeight;
                  boxes[compareBox] = leftWeight;
               }
            }
            else {
               if (rightWeight > leftWeight) {
                  boxes[lightestBoxNum] = rightWeight;
                  boxes[compareBox] = leftWeight;
               }
            }
         }
      }
   }
}
```

You will notice that it runs faster. To get the time, we can write a little program that grabs the CPU time (in milliseconds) before it starts the program, then runs our program, then gets the CPU time again afterwards. We could then determine how long each takes to run:

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

int main() {
      printf("\nC Sort\n");
      system("date +\"%T.%3N\"");
      system("./cSort");
      system("date +\"%T.%3N\"");

      printf("\nC Register Sort\n");
      system("date +\"%T.%3N\"");
      system("./rSort");
      system("date +\"%T.%3N\"");

      printf("\nJava Sort\n");
      system("date +\"%T.%3N\"");
      system("java JSort");
      system("date +\"%T.%3N\"");
}
```

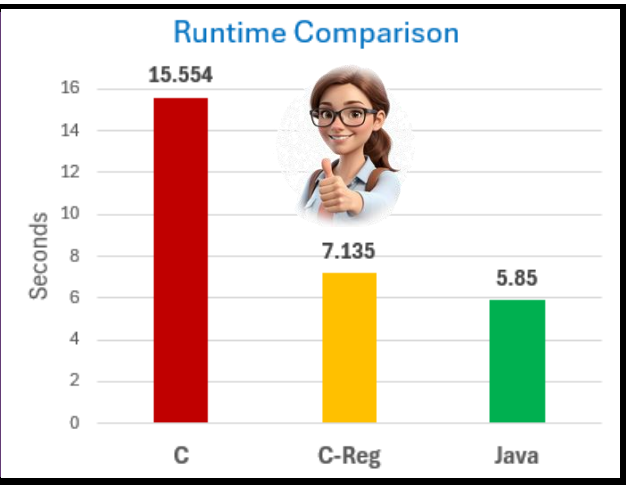
Here is the output (which varies a little each time). Notice the speed chart for comparison:

```
student:~$  ./test
C Sort
14:35:08.189
14:35:23.743

C Register Sort
14:35:23.746
14:35:30.881

Java Sort
14:35:30.884
14:35:36.734

student:~$
```



So … we more than doubled the speed of our program simply by putting the variables in registers! But why is JAVA still faster? Well, there are more optimizations that can be done.

One way to make things more efficient is to write our program in assembly code. Instead of allowing the C compiler to convert to assembly code for us … lets convert our C code into our own assembly code so that we have full control of how to do everything.

We will use **NASM** (a.k.a. Netwide Assembler). You can check is NASM is installed as follows:

```
student:~$  whereis nasm
nasm: /usr/bin/nasm /usr/share/man/man1/nasm.1.gz
student:~$
```

If it was not installed, you would not see the directory that it was installed into. To install, just do this:

```
student:~$  sudo apt-get install nasm
```

You should see it install. It is not the purpose of this course to teach assembly programming. However, we will look at a few assembly commands that we can make use of to create our program. To begin, however, we need to know how to access and use the registers. Here are the registers available to us →

64-bit long	32-bit int	16-bit short	8-bit char
rax	eax	ax	ah/al
rcx	ecx	cx	ch/cl
rdx	edx	dx	dh/dl
rbx	ebx	bx	bh/bl
rsp	esp	sp	spl
rbp	ebp	bp	bpl
rsi	esi	si	sil
rdi	edi	di	dil
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

Notice that there are 16 registers, each being 64-bits in size. There are different names of the registers to allow us to use them as either a **long**, **int**, **short** or **byte**. To write our code, we will also need to know a few assembly commands:

```
mov r8,1000     ;copy val 1000 into reg 8
mov r8,r9       ;copy val from reg 9 into reg 8
mov r8,a[r9]    ;copy r9th val from array a into reg 8
inc r8          ;increment val in reg r8
dec r8          ;decrement val in reg r8
and r8,1        ;bitwise & val in reg 8 with val 1
cmp r8,0        ;compare val in reg 8 with 0
cmp r8,r9       ;compare val in reg 8 with val in reg 9
jne label       ;if most recent comparison was
jge label       ;    !=, >=, <=, <
jle label       ;    then jump to address
jl  label       ;    defined by label
jmp label       ; jump to address defined by label
```

Also, we can declare our boxes array in a section called **.data** as follows:

```
section .data
   boxes  db   "ZYXWVUTSRQPONMLKJIHGFEDCBA",10,0
```

This defines a sequence of bytes in memory (db means *bytes of data*) called boxes. A **10** and **0** are not necessary but they represent the '\n' and NULL-terminator values in case we want to print out the data while debugging.

The assembly code is shown on the next page. Beside it, is pseudocode that corresponds to our algorithm. However, the FOR loops were switched to DO/WHILE loops in the pseudocode since that is closer to what we will be producing in our assembly code.

You will notice that there is a **.data** section and a **.text** section. The **.data** section contains the global variables that we want to define in memory. Our code must have the **_start** label since this is what is looked for (like **main**()) by the assembly linker. There are other labels in the code that represent locations to "jump to" at various times in the program. Finally, the semicolon character allows us to write comments beside our lines of code.

Code from **aSort.asm**

```
section .data
      boxes    db   "ZYXWVUTSRQPONMLKJIHGFEDCBA", 10, 0
                              ;char boxes[26] = {"ZYXWVUTSRQPONMLKJIHGFEDCBA"};

section .text
      global _start

_start:
      mov   r8, 20000000      ; repeat = 20,000,000
do1:                          ; do {
      mov   r9, 0             ;   lightestBoxNum = 0;
do2:                          ;   do {
      mov   r10, r9           ;     compareBox = lightestBoxNum + 1;
      inc   r10               ;
do3:                          ;     do {
      mov   r11b,boxes[r9]    ;       leftWeight = boxes[lightestBoxNum];
      mov   r12b,boxes[r10]   ;       rightWeight = boxes[compareBox];
      mov   r13,r8            ;       if ((repeat & 1) == 0) {
      and   r13,1             ;
      cmp   r13,0             ;
      jne   else              ;
      cmp   r12b,r11b         ;         if (rightWeight < leftWeight) {
      jge   outIf             ;
      mov   boxes[r9],r12b    ;           boxes[lightestBoxNum] = rightWeight;
      mov   boxes[r10],r11b   ;           boxes[compareBox] = leftWeight;
      jmp   outIf             ;         }
                              ;       }
else:                         ;       else {
      cmp   r12b,r11b         ;         if (rightWeight > leftWeight) {
      jle   outIf             ;
      mov   boxes[r9],r12b    ;           boxes[lightestBoxNum] = rightWeight;
      mov   boxes[r10],r11b   ;           boxes[compareBox] = leftWeight;
                              ;         }
outIf:                        ;       }
      inc   r10               ;       compareBox++;
      cmp   r10,26            ;     } while (compareBox < 26);
      jl    do3               ;
      inc   r9                ;     lightestBoxNum++;
      cmp   r9,25             ;   } while (lightestBoxNum < 25);
      jl    do2               ;
      dec   r8                ;   repeat--;
      cmp   r8,0              ; } while (repeat > 0);
      jne   do1               ;
      mov   rax, 60           ; exit(0)
      mov   rdi, 0            ;
      syscall                 ;
```

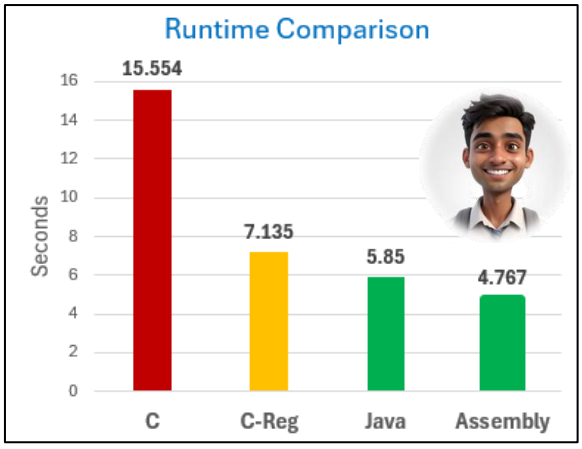
In the code, register **r8** stores the **repeat** variable, **r9** stores the **lightestBoxNum** variable, **r10** stores the **compareBox** variable, **r11b** stores the **leftWeight** variable, **r12b** stores the **rightWeight** variable and **r13** is just used as a temporary variable. Note that it is important that **r11b** and **r12b** are just 8-bit variables because they are storing the ASCII character and when we do a comparison, we only want to compare those **8** bits in the register while ignoring the highest **56** bits of the register.

To compile, link and run the code, we need to do the following:

```
 student:~$  nasm -f elf64 aSort.asm
student:~$  ld -m elf_x86_64 aSort.o -o aSort
student:~$  ./aSort
```

We can then add this to our testing code to see how fast it runs:

```
printf("\nAssembly Sort\n");
system("date +\"%T.%3N\"");
system("./aSort");
system("date +\"%T.%3N\"");
```



As it turns out … we now beat the JAVA runtime. This makes a lot of sense because the machine code produced from the assembly code that we wrote runs directly onto the CPU, whereas JAVA byte code still needs to be interpreted.

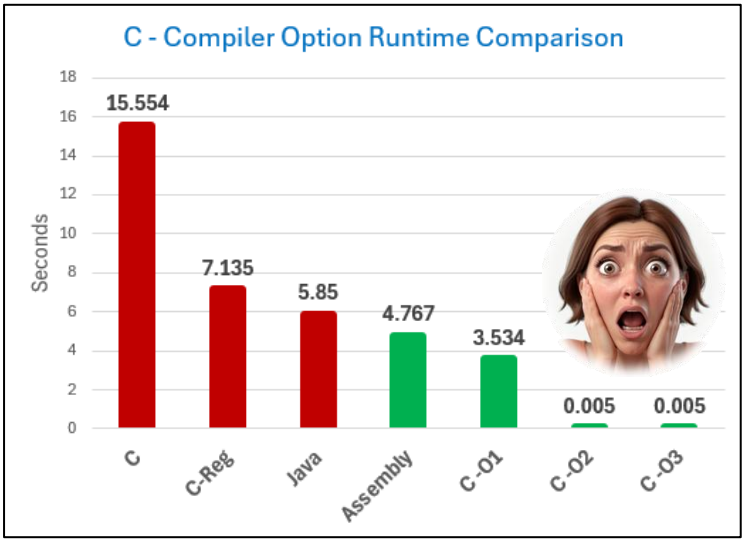
We can improve our code even more … but we get the point. When doing assembly coding, fast code requires efficient use of registers … knowing just *when* and *how long* to keep data in a register. So, doing this can speed up your program … however … it is often better to rely on the compiler and assembler to optimize such things. In fact, the **gcc** compiler also has various optimization capabilities that make writing assembly code unnecessary.

Get ready to blow your mind. Type in the following compile commands (take note that the -O1 is using a “capital Oh” character, not a “zero” character):

```
student:~$  gcc -o rSort-O1 rSort.c -O1
student:~$  gcc -o rSort-O2 rSort.c -O2
student:~$  gcc -o rSort-O3 rSort.c -O3
student:~$
```

This produced three more executables that take advantage of **gcc**’s optimization capabilities in three different levels. We could check the runtime of these by adding more lines to our test code like this:

```
printf("\nC Register O1 Sort\n");
system("date +\"%T.%3N\"");
system("./rSort-O1");
system("date +\"%T.%3N\"");
```



What? How is it possible for the code to be that much faster? The **gcc** optimization is over **3000x** faster than regular **gcc** compiling!!!!!

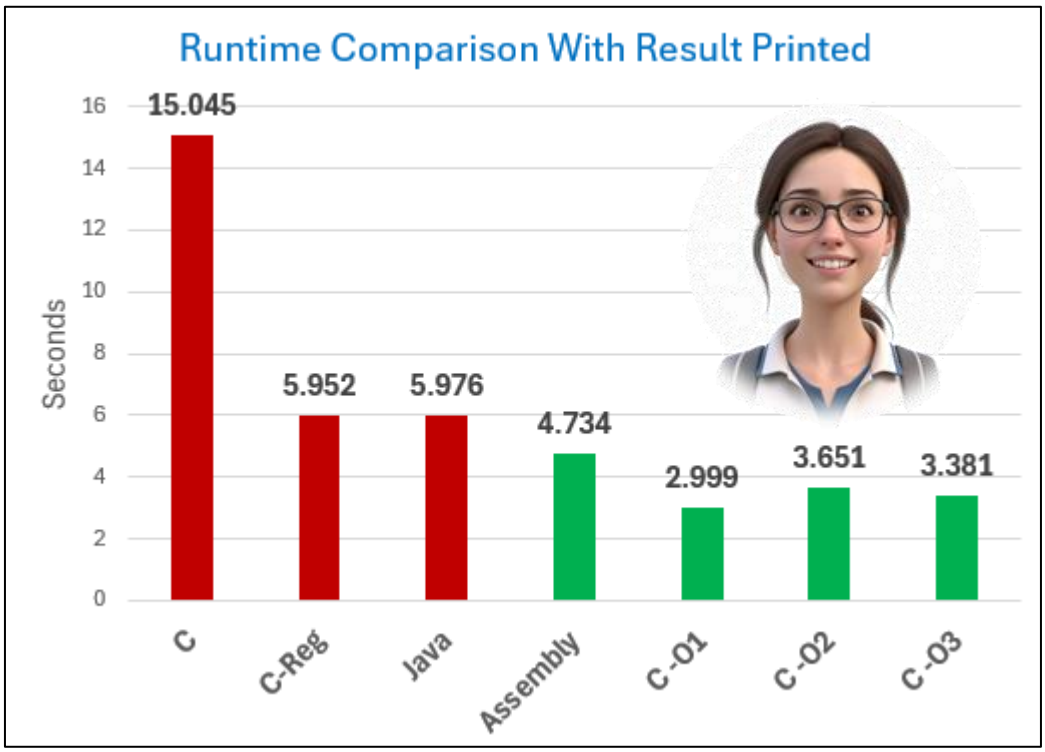
Why would anyone NOT use the **-O3** option?
Why would anyone write assembly code? What is going on here?

If you want to know more about this, you should take the **COMP 3002 - Compiler Construction** course. For now, just realize that there are a lot of "tricks" that we can play. For example, our code doesn't actually do anything because it does not produce any output and it only alters the same **26** bytes of memory. It is a pointless program with no output. The optimization stages of the **gcc** compiler identifies this fact and essentially replaces our loops with no code because it realizes that the end result of the program is the same. So, it is a trick that makes the code look like it is running blindingly fast.

If we simply insert the following lines at the end of the program, then this "trick cannot be played":

```
for (int i=0; i<26; i++)
     printf("%c", boxes[i]);
printf("\n");
```

The end result is that the compiler optimization is no longer able to throw out the FOR loops because our program now needs to produce a result. So, we have a more realistic comparison of the optimization results now as show here:



So, what can we conclude from all of this?

1. By using **registers**, we can speed up our code significantly.
2. By wring **assembly code**, we can obtain better performance than standard compilation.
3. The **gcc** compiler optimization flags may make it unnecessary to write our own assembly code.