Chapter 4

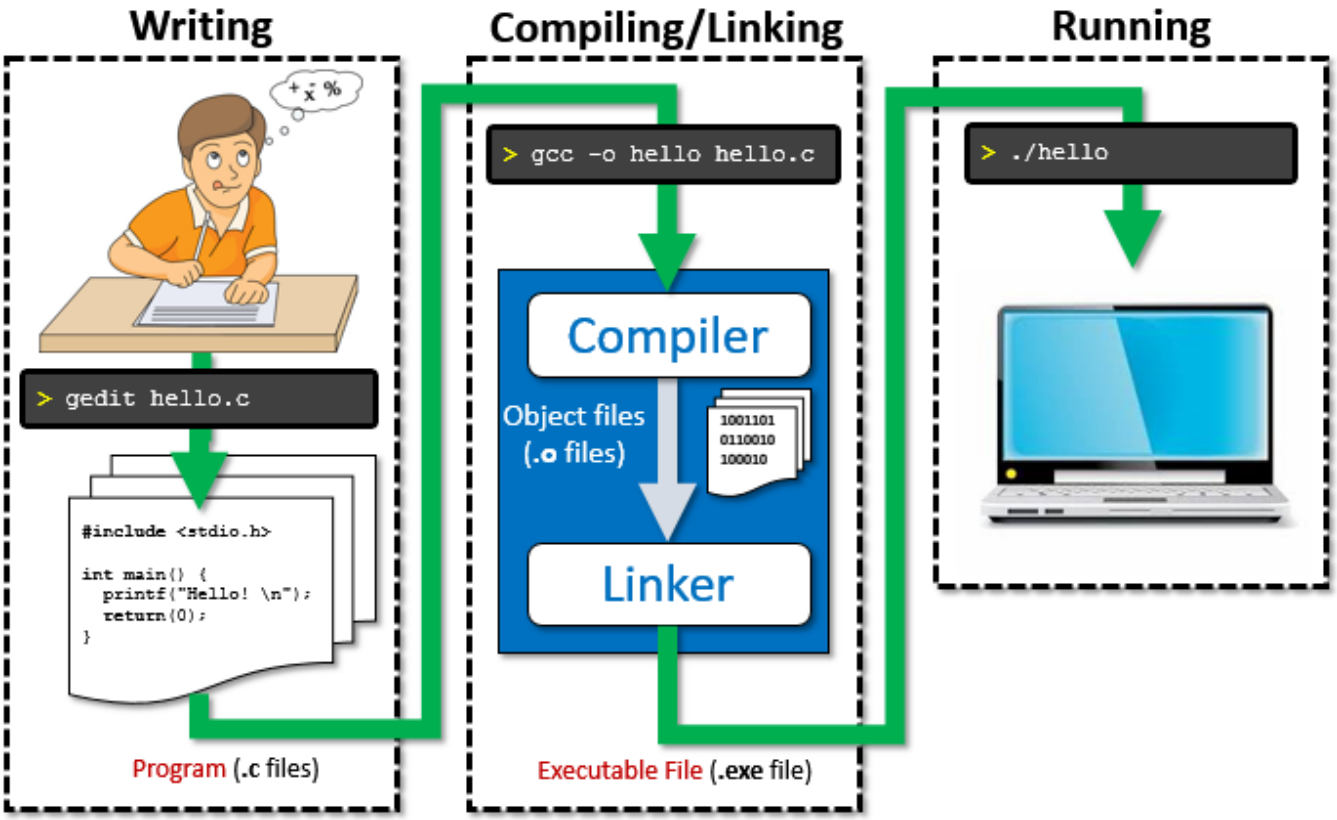
Builds and Makefiles

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**. Finally, it explains how to create simple **makefiles** in order to simplify the process of creating our C programs.

4.1 The Compilation Process

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

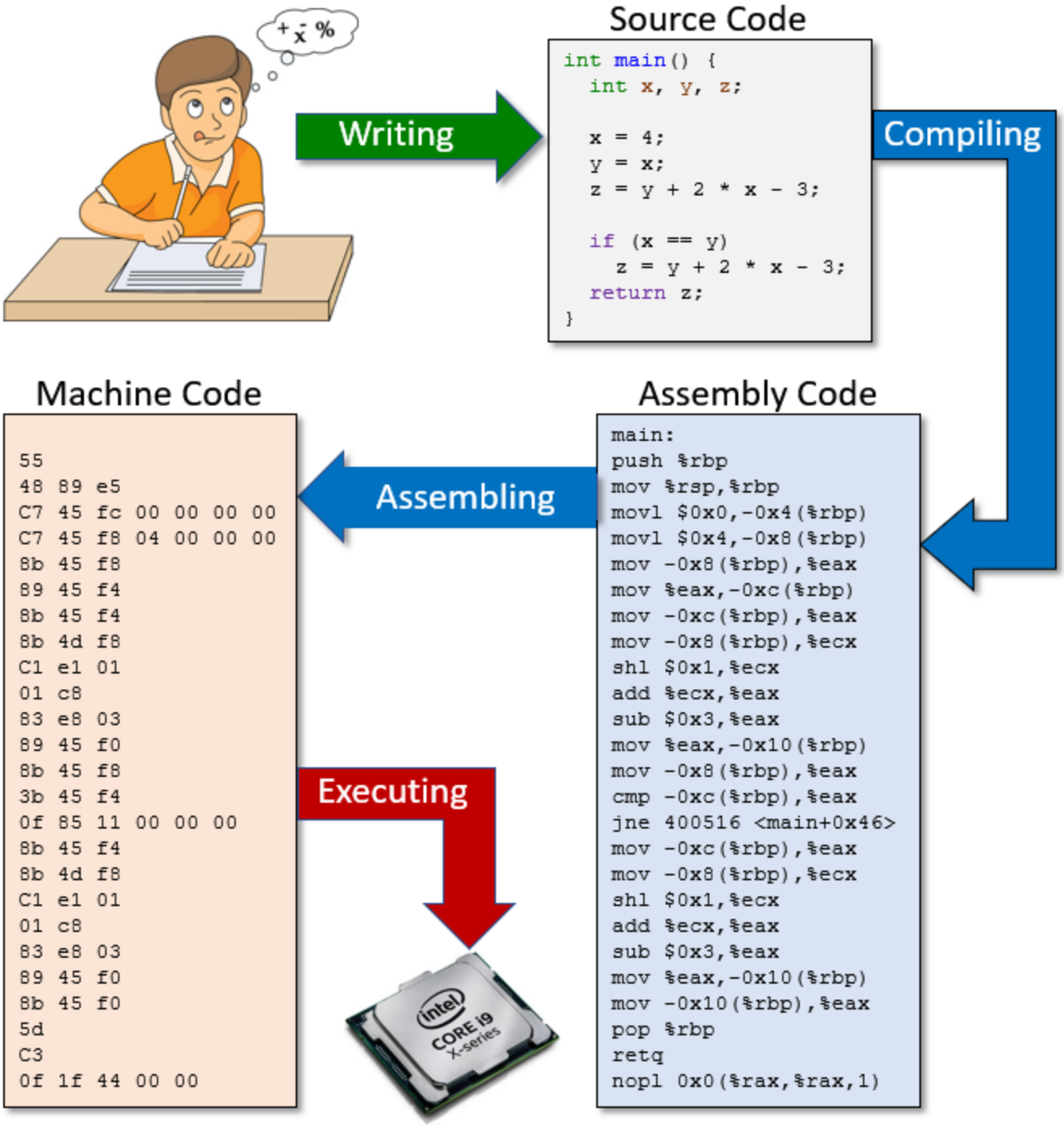


We are going to discuss a little bit more about what happens in the blue box above. That is, we will talk about how the compiler takes your written code and produces an executable file.

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.

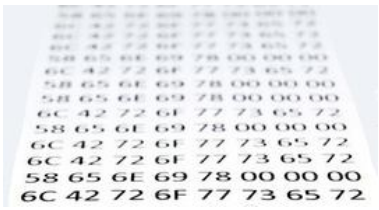
Modern CPUs are Microprocessors … in that all the work happens on a single integrated circuit (IC). Most microprocessors these days have multiple cores … that is … multiple processors (i.e., CPUs) running at the same time in order to process the instructions more quickly (i.e., "many hands make light 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:

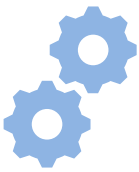


The machine code is very low level and is in a format that the CPU understands. It is really just a set of numbers that represent specific CPU instructions and the accompanying data required for that instruction. The specific instruction codes will be different for each CPU and Operating System. Therefore, 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 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'll 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**	Output

`#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);`
`}`
	`*y = 15`
	
	`x + *y = 30`
	`Days = 7, PI = 3.141590`

`#include <stdio.h>`
`#include "myDefinitions"`
 	
`// #define DEBUG 1` (Commented out now.)
 	
`int main() {`
`  int  x = MAX;`
`  int *y = &x;`
 	
`  #ifdef DEBUG`
`    printf("*y = %d\n", *y);` (Not printed now.)
`  #endif`
`  printf("x + *y = %d\n", x + *y);`
`  printf("Days = %d, PI = %f\n", DAYS_OF_WEEK, PI);`
`}`
	`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 4004 0000 0000 0000  ........@.......
00000030: 0000 0000 4000 0000 0000 4000 0d00 0c00  ....@.....@.....
00000040: 5548 89e5 4883 ec30 6448 8b04 2528 0000  UH..H..0dH..%(..
00000050: 0048 8945 f831 c0c7 45ec 0f00 0000 488d  .H.E.1..E.....H.
00000060: 45ec 4889 45f0 488b 45f0 8b00 89c6 488d  E.H.E.H.E.....H.
00000070: 3d00 0000 00b8 0000 0000 e800 0000 0048  =..............H
00000080: 8b45 f08b 108b 45ec 01d0 89c6 488d 3d00  .E....E.....H.=.
00000090: 0000 00b8 0000 0000 e800 0000 0048 8b05  .............H..
000000a0: 0000 0000 4889 45d8 f20f 1045 d8be 0700  ....H.E....E....
000000b0: 0000 488d 3d00 0000 00b8 0100 0000 e800  ..H.=...........
000000c0: 0000 00b8 0000 0000 488b 4df8 6448 330c  ........H.M.dH3.
000000d0: 2528 0000 0074 05e8 0000 0000 c9c3 0000  %(...t..........
000000e0: 2a79 203d 2025 640a 0078 202b 202a 7920  *y = %d..x + *y
000000f0: 3d20 2564 0a00 4461 7973 203d 2025 642c  = %d..Days = %d,
00000100: 2050 4920 3d20 2566 0a00 0000 0000 0000   PI = %f........
00000110: 6e86 1bf0 f921 0940 0047 4343 3a20 2855  n....!.@.GCC: (U
00000120: 6275 6e74 7520 372e 332e 302d 3237 7562  buntu 7.3.0-27ub
00000130: 756e 7475 317e 3138 2e30 3429 2037 2e33  untu1~18.04) 7.3
00000140: 2e30 0000 0000 0000 1400 0000 0000 0000  .0..............
00000150: 017a 5200 0178 1001 1b0c 0708 9001 0000  .zR..x..........
00000160: 1c00 0000 1c00 0000 0000 0000 9e00 0000  ................
00000170: 0041 0e10 8602 430d 0602 990c 0708 0000  .A....C.........
00000180: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000190: 0000 0000 0000 0000 0100 0000 0400 f1ff  ................
000001a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001b0: 0000 0000 0300 0100 0000 0000 0000 0000  ................
000001c0: 0000 0000 0000 0000 0000 0000 0300 0300  ................
000001d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001e0: 0000 0000 0300 0400 0000 0000 0000 0000  ................
000001f0: 0000 0000 0000 0000 0000 0000 0300 0500  ................
00000200: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000210: 0000 0000 0300 0700 0000 0000 0000 0000  ................
00000220: 0000 0000 0000 0000 0000 0000 0300 0800  ................
00000230: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000240: 0000 0000 0300 0600 0000 0000 0000 0000  ................
00000250: 0000 0000 0000 0000 1000 0000 1200 0100  ................
00000260: 0000 0000 0000 0000 9e00 0000 0000 0000  ................
00000270: 1500 0000 1000 0000 0000 0000 0000 0000  ................
00000280: 0000 0000 0000 0000 2b00 0000 1000 0000  ........+.......
00000290: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000002a0: 3200 0000 1000 0000 0000 0000 0000 0000  2...............
000002b0: 0000 0000 0000 0000 0069 6664 6566 4578  .........ifdefEx
000002c0: 616d 706c 652e 6300 6d61 696e 005f 474c  ample.c.main._GL
000002d0: 4f42 414c 5f4f 4646 5345 545f 5441 424c  OBAL_OFFSET_TABL
000002e0: 455f 0070 7269 6e74 6600 5f5f 7374 6163  E_.printf.__stac
000002f0: 6b5f 6368 6b5f 6661 696c 0000 0000 0000  k_chk_fail......
00000300: 3100 0000 0000 0000 0200 0000 0500 0000  1...............
00000310: fcff ffff ffff ffff 3b00 0000 0000 0000  ........;.......
00000320: 0400 0000 0b00 0000 fcff ffff ffff ffff  ................
00000330: 4f00 0000 0000 0000 0200 0000 0500 0000  O...............
00000340: 0500 0000 0000 0000 5900 0000 0000 0000  ........Y.......
00000350: 0400 0000 0b00 0000 fcff ffff ffff ffff  ................
00000360: 6000 0000 0000 0000 0200 0000 0500 0000  `...............
00000370: 2c00 0000 0000 0000 7500 0000 0000 0000  ,.......u.......
00000380: 0200 0000 0500 0000 1200 0000 0000 0000  ................
00000390: 7f00 0000 0000 0000 0400 0000 0b00 0000  ................
000003a0: fcff ffff ffff ffff 9800 0000 0000 0000  ................
000003b0: 0400 0000 0c00 0000 fcff ffff ffff ffff  ................
000003c0: 2000 0000 0000 0000 0200 0000 0200 0000   ...............
000003d0: 0000 0000 0000 0000 002e 7379 6d74 6162  ..........symtab
000003e0: 002e 7374 7274 6162 002e 7368 7374 7274  ..strtab..shstrt
000003f0: 6162 002e 7265 6c61 2e74 6578 7400 2e64  ab..rela.text..d
00000400: 6174 6100 2e62 7373 002e 726f 6461 7461  ata..bss..rodata
00000410: 002e 636f 6d6d 656e 7400 2e6e 6f74 652e  ..comment..note.
00000420: 474e 552d 7374 6163 6b00 2e72 656c 612e  GNU-stack..rela.
00000430: 6568 5f66 7261 6d65 0000 0000 0000 0000  eh_frame........
00000440: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000450: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000460: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000470: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000480: 2000 0000 0100 0000 0600 0000 0000 0000   ...............
00000490: 0000 0000 0000 0000 4000 0000 0000 0000  ........@.......
000004a0: 9e00 0000 0000 0000 0000 0000 0000 0000  ................
000004b0: 0100 0000 0000 0000 0000 0000 0000 0000  ................
000004c0: 1b00 0000 0400 0000 4000 0000 0000 0000  ........@.......
000004d0: 0000 0000 0000 0000 0003 0000 0000 0000  ................
000004e0: c000 0000 0000 0000 0a00 0000 0100 0000  ................
000004f0: 0800 0000 0000 0000 1800 0000 0000 0000  ................
00000500: 2600 0000 0100 0000 0300 0000 0000 0000  &...............
00000510: 0000 0000 0000 0000 de00 0000 0000 0000  ................
00000520: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000530: 0100 0000 0000 0000 0000 0000 0000 0000  ................
00000540: 2c00 0000 0800 0000 0300 0000 0000 0000  ,...............
00000550: 0000 0000 0000 0000 de00 0000 0000 0000  ................
```

```
00000560: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000570: 0100 0000 0000 0000 0000 0000 0000 0000  ................
00000580: 3100 0000 0100 0000 0200 0000 0000 0000  1...............
00000590: 0000 0000 0000 0000 e000 0000 0000 0000  ................
000005a0: 3800 0000 0000 0000 0000 0000 0000 0000  8...............
000005b0: 0800 0000 0000 0000 0000 0000 0000 0000  ................
000005c0: 3900 0000 0100 0000 3000 0000 0000 0000  9.......0.......
000005d0: 0000 0000 0000 0000 1801 0000 0000 0000  ................
000005e0: 2b00 0000 0000 0000 0000 0000 0000 0000  +...............
000005f0: 0100 0000 0000 0000 0100 0000 0000 0000  ................
00000600: 4200 0000 0100 0000 0000 0000 0000 0000  B...............
00000610: 0000 0000 0000 0000 4301 0000 0000 0000  ........C.......
00000620: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000630: 0100 0000 0000 0000 0000 0000 0000 0000  ................
00000640: 5700 0000 0100 0000 0200 0000 0000 0000  W...............
00000650: 0000 0000 0000 0000 4801 0000 0000 0000  ........H.......
00000660: 3800 0000 0000 0000 0000 0000 0000 0000  8...............
00000670: 0800 0000 0000 0000 0000 0000 0000 0000  ................
00000680: 5200 0000 0400 0000 4000 0000 0000 0000  R.......@.......
00000690: 0000 0000 0000 0000 c003 0000 0000 0000  ................
000006a0: 1800 0000 0000 0000 0a00 0000 0800 0000  ................
000006b0: 0800 0000 0000 0000 1800 0000 0000 0000  ................
000006c0: 0100 0000 0200 0000 0000 0000 0000 0000  ................
000006d0: 0000 0000 0000 0000 8001 0000 0000 0000  ................
000006e0: 3801 0000 0000 0000 0b00 0000 0900 0000  8...............
000006f0: 0800 0000 0000 0000 1800 0000 0000 0000  ................
00000700: 0900 0000 0300 0000 0000 0000 0000 0000  ................
00000710: 0000 0000 0000 0000 b802 0000 0000 0000  ................
00000720: 4300 0000 0000 0000 0000 0000 0000 0000  C...............
00000730: 0100 0000 0000 0000 0000 0000 0000 0000  ................
00000740: 1100 0000 0300 0000 0000 0000 0000 0000  ................
00000750: 0000 0000 0000 0000 d803 0000 0000 0000  ................
00000760: 6100 0000 0000 0000 0000 0000 0000 0000  a...............
00000770: 0100 0000 0000 0000 0000 0000 0000 0000  ................
```

The left side shows the byte numbers (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. Any time that we **#include** a header file, we are usually doing that 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:  main util
      gcc -o linkExample linkExampleMain.o linkExampleUtil.o

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

util: 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., **main** & **util**). They are arbitrary labels, but it is good to have them somewhat match up logically with what is happening at that point in the makefile.

The first line represents the dependencies for the executable. That is, by means of the two labels **main** and **util**, 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 in order 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. As long as 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
cc    -c -o linkExampleMain.o linkExampleMain.c
cc    -c -o linkExampleUtil.o 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 the 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: main util
	gcc -o linkExample $(OBJ)

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

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

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

```
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 **main** file. We could just specify the **main** label in the make command call like this:

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

Or if we just want to compile the **linkExampleMain.c** and **linkExampleUtil.c** files without linking, we can specify both makefile labels like this:

```
student@COMPBase:~$ make main util
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 cleanup 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: main util
      gcc -o linkExample $(OBJ)

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

util: 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
cc    -c -o linkExampleMain.o linkExampleMain.c
cc    -c -o linkExampleUtil.o 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.

Make files can get quite large and even complicated. Here is an old makefile of mine with a few things that I'd 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
```

Comments can be made in a makefile by using the # character.

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'd just have to make the substitution.

```
#########################################################
# 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


#########################################################
# 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
```

It is nice to make variables for various libraries.

Use a backslash character to indicate that the stuff on the next line is supposed to be appended here.

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

```
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
```

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