CMPUT201W20B2 Week 10

Abram Hindle

March 17, 2020

Contents

1 Week10

1.1 Copyright Statement

If you are in CMPUT201 at UAlberta this code is released in the public domain to you.

1.1.1 License

1.1.2 Hazel Code is licensed under AGPL3.0+

Hazel's code is also found here `https://github.com/hazelybell/examples/tree/C-2020-01`

Hazel code is licensed: The example code is licensed under the AGPL3+ license, unless otherwise noted.

1.2 Init ORG-MODE

```
;; I need this for org-mode to work well
;; If we have a new org-mode use ob-shell
;; otherwise use ob-sh --- but not both!
(if (require 'ob-shell nil 'noerror)
  (progn
    (org-babel-do-load-languages 'org-babel-load-languages '((shell . t))))
  (progn
    (require 'ob-sh)
    (org-babel-do-load-languages 'org-babel-load-languages '((sh . t)))))
(org-babel-do-load-languages 'org-babel-load-languages '((C . t)))
(org-babel-do-load-languages 'org-babel-load-languages '((python . t)))
(setq org-src-fontify-natively t)
(setq org-confirm-babel-evaluate nil) ;; danger!
(custom-set-faces
 '(org-block ((t (:inherit shadow :foreground "black"))))
 '(org-code ((t (:inherit shadow :foreground "black")))))
```

1.2.1 Org export

```
(org-html-export-to-html)
(org-latex-export-to-pdf)
(org-ascii-export-to-ascii)
```

1.2.2 Org Template

Copy and paste this to demo C

```
#include <stdio.h>

int main(int argc, char**argv) {
    return 0;
}
```

1.3 Remember how to compile?

gcc -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -o programname programname.c

1.4 Preprocessor stuff like if-def

The preprocessor deals with all the lines that you start with an octalthrope or hash mark: #

The preprocessor lets you define symbols, macros, and include files.

1.4.1 Multiple Files?

How does stdio.h work?

`file:///usr/include/stdio.h`

It defines definitions, macros, and prototypes for the stdio library. The linker will link your executable to that library that was already compiled.

.h files help us organize C programs by including definitions for the object files and libraries that we will create.

Libc or glibc contains the implemention of those definitions. libc.so.6 => /lib/x86$_{64}$-linux-gnu/libc.so.6 (0x00007f919f994000)

libc is composed of many .c files compiled into .o object files and then combined into a library. A library is like an executable that other executables rely on for code. malloc is defined in malloc.c and has a malloc.h file!

Typically if I make a library I will make a .h file so the definitions can be shared with other .c files. But the implementation of the functions will go into a .c file that includes that .h as well.

- main.c
  - #include "library.h"
  - relies on library.o
- library.c
  - #include "library.h"
  - makes library.o
- library.h
  - defines functions and definitions from library.c

1.4.2 Example

This is a useful function to check if scanf read 1 or more elements and didn't read EOF.

`./checkinput.c`

```
#include "checkinput.h"
#include <stdio.h>
#include <stdlib.h>
/* checkInput: given the result of scanf check if it
 * 0 elements read or EOF. If so exit(1) with a warning.
 *
 */
void checkInput(int err) {
  if (!err || err == EOF) {
    printf("\nInvalid input!\n");
    exit(1);
  }
}
```

`./checkinput.h`

```
// Have a guard to ensure that we don't include it multiple times.
#ifndef _CHECKINPUT_H_
/* checkInput: given the result of scanf check if it
 * 0 elements read or EOF. If so exit(1) with a warning.
 *
 */
#define _CHECKINPUT_H_
void checkInput(int err); // a prototype!
#endif
```

./checkinput-driver.c

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

int main() {
  int input;
  checkInput(scanf("%d", &input));
  puts("Good Input!");
}
```

1. Compiling Multiple Files Easy Mode

   We can put all our .c files on the same line and compile them all at once! This is handy. But quite limiting.

   We can't parallel compile. We can't use all our cores. We can't interupt compilation.

   ```
   # build checkinput-driver
   gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
          -o checkinput-driver checkinput-driver.c \
          checkinput.c && \
   ( echo YES | ./checkinput-driver  || \
     echo 100 | ./checkinput-driver )
   ```

   ```
   Invalid input!
   Good Input!
   ```

2. Compiling Multiple Files with Linking

   OK now we compile it. The main is the last to compile and it needs all the .o files.

   All the .c files that don't contain main need to be compiled to object files. Use the -c flags to do this.

```
# build checkinput.o
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
       -c checkinput.c
file checkinput.o
# build checkinput-driver and link it to checkinput.o
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
       -o checkinput-driver checkinput-driver.c \
       checkinput.o
file checkinput-driver
```

```
checkinput.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), with debug_inf
checkinput-driver: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamic
```

   - If you want some translation :-)
     - ELF - Executable and Linking format
     - relocatable - you can link it
     - shared object - relocatable and executable
     - LSB - little endian/least significant bit
     - x86-64 - 64 bit x86 processor
     - version 1 (SYSV) - version 1 of ELF System V Unix spec.

   Test drive it

```
echo    | ./checkinput-driver # bad
echo X  | ./checkinput-driver # bad
echo 1  | ./checkinput-driver # good
echo -1 | ./checkinput-driver # good
```

```

Invalid input!

Invalid input!
Good Input!
Good Input!
```

Now let's see how it is linked!

```
ldd ./checkinput-driver
```

```
linux-vdso.so.1 (0x00007ffe85be0000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f919f994000)
/lib64/ld-linux-x86-64.so.2 (0x00007f919ff87000)
```

- syscalls (read, write, gettimeofday) and libc (libc is stuff like stdio.h)

1.4.3 Linking to libraries

'math.h' includes fun functions like cos and tanh.

Math.h, part of the C stdlib, is distributed as a seperate library. Not all computers have floating point numbers so why bother compiling floating code for them?

`file:///usr/include/math.h`

```
gnome-terminal --window-with-profile Big \
               -- man math.h
```

I add the flag -lm so we get our math library :-)

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

int main() {
    double x = 0.0;
    double th = tanh(x);
    double lh = th;
    do {
        lh = th;
        x += 0.5;
        th = tanh(x);
        printf("tanh(%e) == %e\n", x, th);
    } while( lh != th );
}
```

```
tanh(5.000000e-01) == 4.621172e-01
tanh(1.000000e+00) == 7.615942e-01
tanh(1.500000e+00) == 9.051483e-01
tanh(2.000000e+00) == 9.640276e-01
tanh(2.500000e+00) == 9.866143e-01
tanh(3.000000e+00) == 9.950548e-01
tanh(3.500000e+00) == 9.981779e-01
tanh(4.000000e+00) == 9.993293e-01
tanh(4.500000e+00) == 9.997532e-01
tanh(5.000000e+00) == 9.999092e-01
tanh(5.500000e+00) == 9.999666e-01
tanh(6.000000e+00) == 9.999877e-01
tanh(6.500000e+00) == 9.999955e-01
tanh(7.000000e+00) == 9.999983e-01
tanh(7.500000e+00) == 9.999994e-01
tanh(8.000000e+00) == 9.999998e-01
tanh(8.500000e+00) == 9.999999e-01
tanh(9.000000e+00) == 1.000000e+00
tanh(9.500000e+00) == 1.000000e+00
tanh(1.000000e+01) == 1.000000e+00
tanh(1.050000e+01) == 1.000000e+00
tanh(1.100000e+01) == 1.000000e+00
tanh(1.150000e+01) == 1.000000e+00
tanh(1.200000e+01) == 1.000000e+00
tanh(1.250000e+01) == 1.000000e+00
tanh(1.300000e+01) == 1.000000e+00
tanh(1.350000e+01) == 1.000000e+00
tanh(1.400000e+01) == 1.000000e+00
tanh(1.450000e+01) == 1.000000e+00
tanh(1.500000e+01) == 1.000000e+00
tanh(1.550000e+01) == 1.000000e+00
tanh(1.600000e+01) == 1.000000e+00
tanh(1.650000e+01) == 1.000000e+00
tanh(1.700000e+01) == 1.000000e+00
tanh(1.750000e+01) == 1.000000e+00
tanh(1.800000e+01) == 1.000000e+00
tanh(1.850000e+01) == 1.000000e+00
tanh(1.900000e+01) == 1.000000e+00
tanh(1.950000e+01) == 1.000000e+00
tanh(2.000000e+01) == 1.000000e+00
```

OK so how does this work, how do link to math?

```
# build checkinput-driver and link it to checkinput.o
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
       -o poor-tanh-example poor-tanh-example.c \
       -lm
file poor-tanh-example
./poor-tanh-example | wc
ldd ./poor-tanh-example

poor-tanh-example: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
     40     120    1400
linux-vdso.so.1 (0x00007ffdd1570000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f4ed467b000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f4ed428a000)
/lib64/ld-linux-x86-64.so.2 (0x00007f4ed4c1b000)
```

See that? libm.so.6 is in there.

Larger programs link to lots of libraries.

```
ldd ‘which xterm‘

linux-vdso.so.1 (0x00007ffdabcea000)
libXft.so.2 => /usr/lib/x86_64-linux-gnu/libXft.so.2 (0x00007fc90b414000)
libfontconfig.so.1 => /usr/lib/x86_64-linux-gnu/libfontconfig.so.1 (0x00007fc90b1cf000
libXaw.so.7 => /usr/lib/x86_64-linux-gnu/libXaw.so.7 (0x00007fc90af5b000)
libXmu.so.6 => /usr/lib/x86_64-linux-gnu/libXmu.so.6 (0x00007fc90ad42000)
libXt.so.6 => /usr/lib/x86_64-linux-gnu/libXt.so.6 (0x00007fc90aad9000)
libX11.so.6 => /usr/lib/x86_64-linux-gnu/libX11.so.6 (0x00007fc90a7a1000)
libXinerama.so.1 => /usr/lib/x86_64-linux-gnu/libXinerama.so.1 (0x00007fc90a59e000)
libXpm.so.4 => /usr/lib/x86_64-linux-gnu/libXpm.so.4 (0x00007fc90a38c000)
libICE.so.6 => /usr/lib/x86_64-linux-gnu/libICE.so.6 (0x00007fc90a171000)
libutempter.so.0 => /usr/lib/x86_64-linux-gnu/libutempter.so.0 (0x00007fc909f6e000)
libtinfo.so.5 => /lib/x86_64-linux-gnu/libtinfo.so.5 (0x00007fc909d44000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fc909953000)
libfreetype.so.6 => /usr/lib/x86_64-linux-gnu/libfreetype.so.6 (0x00007fc90969f000)
libXrender.so.1 => /usr/lib/x86_64-linux-gnu/libXrender.so.1 (0x00007fc909495000)
libexpat.so.1 => /lib/x86_64-linux-gnu/libexpat.so.1 (0x00007fc909263000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fc909044000)
libXext.so.6 => /usr/lib/x86_64-linux-gnu/libXext.so.6 (0x00007fc908e32000)
libSM.so.6 => /usr/lib/x86_64-linux-gnu/libSM.so.6 (0x00007fc908c2a000)
libxcb.so.1 => /usr/lib/x86_64-linux-gnu/libxcb.so.1 (0x00007fc908a02000)
```

```
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fc9087fe000)
libbsd.so.0 => /lib/x86_64-linux-gnu/libbsd.so.0 (0x00007fc9085e9000)
/lib64/ld-linux-x86-64.so.2 (0x00007fc90b8d9000)
libpng16.so.16 => /usr/lib/x86_64-linux-gnu/libpng16.so.16 (0x00007fc9083b7000)
libz.so.1 => /lib/x86_64-linux-gnu/libz.so.1 (0x00007fc90819a000)
libuuid.so.1 => /lib/x86_64-linux-gnu/libuuid.so.1 (0x00007fc907f93000)
libXau.so.6 => /usr/lib/x86_64-linux-gnu/libXau.so.6 (0x00007fc907d8f000)
libXdmcp.so.6 => /usr/lib/x86_64-linux-gnu/libXdmcp.so.6 (0x00007fc907b89000)
librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007fc907981000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fc9075e3000)
```

See! Lots of libraries!

1. Summary

   To link to a shared library with gcc or clang use the: -l flag -llibraryyouwant

   For libm use -lm for librt use -lrt

   If your library is not in the current lib path you will need to specify a library path use -L/path/to/library

   OK let's see how it affects you.

1.4.4 Example Datastructure

Let's make a brief data structure about one of my favourite topics: cool bears.

`./coolbears.c`

```
#define _POSIX_C_SOURCE 200809L // <-- needed for strdup
#include "coolbears.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// hiding struct details from other programmers
// I DONT TRUST THEM. Especially Hazel ;-) (don't tell hazel)
struct coolbear_t {
    char * name;
    float temperature;
};

CoolBear createCoolBear(char * name, float temperature) {
    CoolBear coolbear = malloc(sizeof(*coolbear));
    coolbear->name = strdup(name);
    coolbear->temperature = temperature;
    return coolbear;
}
void freeCoolBear(CoolBear coolBear) {
    if (coolBear == NULL) {
        abort();
    }
    if (coolBear->name != NULL) {
        free(coolBear->name);
    }
    free(coolBear);
}
char * getNameCoolBear(CoolBear coolbear) {
    return coolbear->name;
}
float    getTemperatureCoolBear(CoolBear coolbear) {
   return coolbear->temperature;
}
// NO MAIN!
```

./coolbears.h

```
// Have a guard to ensure that we don't include it multiple times.
#ifndef _COOLBEARS_H_
/* checkInput: given the result of scanf check if it
 * 0 elements read or EOF. If so exit(1) with a warning.
 *
 */
#define _COOLBEARS_H_
struct coolbear_t; // Forward declaration -- I am not sharing details!
typedef struct coolbear_t * CoolBear; // Struct point as type

CoolBear createCoolBear(char * name, float temperature); // a prototype!
void     freeCoolBear(CoolBear coolBear); // a prototype!
char *   getNameCoolBear(CoolBear coolbear); // a prototype!
float    getTemperatureCoolBear(CoolBear coolbear); // a prototype!

#endif
```

./coolbears-driver.c

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

int main() {
  CoolBear ziggy = createCoolBear("Ziggy",-23.0 /* C */);
  CoolBear kevin = createCoolBear("Kevin",-32.0 /* C */);
  CoolBear coolest = (getTemperatureCoolBear(ziggy) <
                      getTemperatureCoolBear(kevin))? ziggy : kevin;
  printf("The coolest bear is %s\n", getNameCoolBear( coolest ));
  // // we actually don't know about name so we can't reference it below
  // printf("The coolest bear is %s\n", getNameCoolBear( coolest->name ));
  freeCoolBear(ziggy);
  freeCoolBear(kevin);
}
```

Compile it. -c the coolbears.c to make coolbears.o and then compile coolbears-driver.c

coolbears-driver.c has no clue how to access

```
# build coolbears.o
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
       -c coolbears.c
# build coolbears-driver and link it to coolbears.o
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
       -o coolbears-driver coolbears-driver.c \
       coolbears.o
./coolbears-driver
```

```
The coolest bear is Kevin
```

If we access coolest->name we get:

```
coolbears-driver.c: In function ‘main’:
coolbears-driver.c:11:62: error: dereferencing pointer to incomplete type ‘struct cool
   printf("The coolest bear is %s\n", getNameCoolBear( coolest->name ));
```

1.4.5 What is the preprocessor doing?

Let's use the -E flag to see what checkinput.c becomes

This output contains glibc headers for stdio.h and stdlib.h these should be under the GPLV3 (c) the Glibc project and GNU project.

If you want more preprocessor options checkout:

`https://gcc.gnu.org/onlinedocs/gcc-5.2.0/gcc/Preprocessor-Options.html`

```
# build checkinput.o
gcc -E -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
       checkinput.c > checkinput-preprocessor.c
```

It produces this file:

`checkinput-preprocessor.c`

```
# 1 "checkinput.c"
# 1 "/home/hindle1/projects/CMPUT201W20/2020-01/CMPUT201W20B2-public/week10//"
# 1 "<built-in>"
#define __STDC__ 1
#define __STDC_VERSION__ 199901L
#define __STDC_HOSTED__ 1
#define __GNUC__ 7
#define __GNUC_MINOR__ 5
#define __GNUC_PATCHLEVEL__ 0

// lots of definitions

# 1 "/usr/include/stdio.h" 1 3 4
# 24 "/usr/include/stdio.h" 3 4
#define _STDIO_H 1

// Start of STDIO_H

// ...

extern int printf (const char *__restrict __format, ...);

extern int sprintf (char *__restrict __s,
      const char *__restrict __format, ...) __attribute__ ((__nothrow__));


// LOTS OF STDIO.H
// LOTS OF STDLIB.H

# 1016 "/usr/include/stdlib.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/stdlib-float.h" 1 3 4
# 1017 "/usr/include/stdlib.h" 2 3 4
# 1026 "/usr/include/stdlib.h" 3 4

# 5 "checkinput.c" 2






# 9 "checkinput.c"
void checkInput(int err) {
  if (!err || err ==
# 10 "checkinput.c" 3 4
                    (-1)
# 10 "checkinput.c"
                      ) {
    printf("\nInvalid input!\n");
    exit(1);
  }
}
return 0;
}
```

checkinput-preprocessor.c

1.4.6 Parameterized Macros

As we just demonstrated Macros generate code. So we can make compile functions that generate code. These functions run at compile time and generate code that is compiled by C.

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

```
// RELU is a rectified linear unit. These are popular in convolutional neural networks
// they are 0 for 0 and negative numbers and they are the identity for positive number
// RELU(-100) = RELU(-1) = 0 && RELU(1) = 1 && RELU(100) = 100
#define RELU(x)  (( x < 0 )?0:x)

int main() {
    // ints
    for (int i = -10; i < 10; i++) {
        printf("RELU(%d)=%d\n", i, RELU(i));
    }
    puts("\n");
    // more in the range of neural networks
    for (double i = -1; i < 1; i+=0.1) {
        printf("RELU(%f)=%f\n", i, RELU(i));
    }
    puts("\n");
}
```

```
RELU(-10)=0
RELU(-9)=0
RELU(-8)=0
RELU(-7)=0
RELU(-6)=0
RELU(-5)=0
RELU(-4)=0
RELU(-3)=0
RELU(-2)=0
RELU(-1)=0
RELU(0)=0
RELU(1)=1
RELU(2)=2
RELU(3)=3
RELU(4)=4
RELU(5)=5
RELU(6)=6
RELU(7)=7
RELU(8)=8
RELU(9)=9
RELU(-1.000000)=0.000000
RELU(-0.900000)=0.000000
RELU(-0.800000)=0.000000
RELU(-0.700000)=0.000000
RELU(-0.600000)=0.000000
RELU(-0.500000)=0.000000
RELU(-0.400000)=0.000000
RELU(-0.300000)=0.000000
RELU(-0.200000)=0.000000
RELU(-0.100000)=0.000000
RELU(-0.000000)=0.000000
RELU(0.100000)=0.100000
RELU(0.200000)=0.200000
RELU(0.300000)=0.300000
RELU(0.400000)=0.400000
RELU(0.500000)=0.500000
RELU(0.600000)=0.600000
RELU(0.700000)=0.700000
RELU(0.800000)=0.800000
RELU(0.900000)=0.900000
RELU(1.000000)=1.000000
```

```
gcc -E -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
       relu.c > \
       relu-expanded.c
```

relu-expanded.c

```
#define RELU(x) (( x < 0 )?0:x)


# 10 "relu.c"
int main() {

    for (int i = -10; i < 10; i++) {
        printf("RELU(%d)=%d\n", i, (( i < 0 )?0:i));
    }
    puts("\n");

    for (double i = -1; i < 1; i+=0.1) {
        printf("RELU(%f)=%f\n", i, (( i < 0 )?0:i));
    }
    puts("\n");
}
```

That's interesting, but be aware that x is not a value. It is a set of tokens.

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

// RELU is a rectified linear unit. These are popular in convolutional neural networks
// they are 0 for 0 and negative numbers and they are the identity for positive numbers
// RELU(-100) = RELU(-1) = 0 && RELU(1) = 1 && RELU(100) = 100
#define RELU(x)  (( x < 0 )?0:x)

int main() {
    double x = 2.0;
    double y = 127.1;
    // How many times will pow(x,y) run?
    printf("%f\n", RELU(pow(x,y)));

}
```

```
1.8235280531744908e+38
```

```
gcc -E -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
       relu2.c > \
       relu2-expanded.c
```

relu2-expanded.c

```
#define RELU(x) (( x < 0 )?0:x)


# 11 "relu2.c"
int main() {
    double x = 2.0;
    double y = 127.1;
    // Uh oh how many pows?
    printf("%f\n", (( pow(x,y) < 0 )?0:pow(x,y)));

}
```

1. Easy bugs!

   So why doesn't this work?

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

   // if checkinput is true then you have an error
   #define CHECKINPUT(scanfReturn)  ( scanfReturn == EOF || !scanfReturn )

   int main() {
       int myInt = 0;
       if (CHECKINPUT(scanf("%d", &myInt))) {
           printf("Invalid input!");
           exit(1);
       }
       printf("My int: %d\n", myInt);
   }
   ```

   ```
   gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
          -o checkinputmacro checkinputmacro.c
   echo   | ./checkinputmacro
   echo X | ./checkinputmacro
   echo 6 | ./checkinputmacro
   ```

   ```
   gcc -E -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
          checkinputmacro.c > \
          checkinputmacro-expanded.c
   ```

   `checkinputmacro-expanded.c`

   Let's look at the output:

   ```
   #define CHECKINPUT(scanfReturn) ( scanfReturn == false || scanfReturn == EOF)
   ```

```
# 8 "checkinputmacro.c"
int main() {
    int myInt = 0;
    if (( scanf("%d", &myInt) ==
# 10 "checkinputmacro.c" 3 4
       0
# 10 "checkinputmacro.c"
       || scanf("%d", &myInt) ==
# 10 "checkinputmacro.c" 3 4
       (-1)
# 10 "checkinputmacro.c"
       )) {
        printf("Invalid input!");
        exit(1);
    }
    printf("My int: %d\n", myInt);
}
```

I'll clear it up for you

```
#define CHECKINPUT(scanfReturn) ( scanfReturn == false || scanfReturn == EOF)

int main() {
    int myInt = 0;
    if (( scanf("%d", &myInt) ==  0 || scanf("%d", &myInt) == (-1))) {
        printf("Invalid input!");
        exit(1);
    }
    printf("My int: %d\n", myInt);
}
```

See? 2 scanfs instead of 1. Great. So macros will copy your tokens, not your values. They are meta-functions and not real functions.

How do we fix? We assign the result once

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

```
// if checkinput is true then you have an error
// horrible and bad style don't do this at home!
static int __ret;
#define CHECKINPUT(scanfReturn)  (__ret = scanfReturn, (__ret== EOF || !__ret ))

int main() {
    int myInt = 0;
    if (CHECKINPUT(scanf("%d", &myInt))) {
        printf("Invalid input!\n");
        exit(1);
    }
    printf("My int: %d\n", myInt);
}
```

```
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
       -o checkinputmacro-fixed checkinputmacro-fixed.c
echo   | ./checkinputmacro-fixed
echo X | ./checkinputmacro-fixed
echo 6 | ./checkinputmacro-fixed
```

```
Invalid input!
Invalid input!
My int: 6
```

```
gcc -E -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
       checkinputmacro-fixed.c > \
       checkinputmacro-fixed-expanded.c
```

checkinputmacro-fixed-expanded.c

```
# 7 "checkinputmacro-fixed.c"
static int __ret;
#define CHECKINPUT(scanfReturn) (__ret = scanfReturn, (__ret== EOF || !__ret ))

int main() {
    int myInt = 0;
    if ((__ret = scanf("%d", &myInt), (__ret==
# 12 "checkinputmacro-fixed.c" 3 4
       (-1)
# 12 "checkinputmacro-fixed.c"
       || !__ret ))) {
        printf("Invalid input!\n");
        exit(1);
    }
    printf("My int: %d\n", myInt);
}
```

1.5 Makefiles

I am sick to death of all these shell scripts!

Look at the assignments 1 shell script per question and they mostly say the same things.

Programmers uses build systems to manage compiling and linking large programs. They often do not use shell scripts or batch files directly.

Makefiles allow you to use make to build your program. Make is declarative, dependency based build system.

Makefiles are full of rules for building files.

```
file-you-want-to-build: dependency1.c dependency2.o dependency3.o
        gcc -o file-you-want-to-build dependency1.c dependency2.o dependency3.o
```

^^^ There is a tab character before the gcc

To build file-you-want-to-build you type:

make file-you-want-to-build

make

1.5.1 Basic Makefile

We're going to use the coolbears source code from before.

Instead of this:

```
# build coolbears.o
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
       -c coolbears.c
# build coolbears-driver and link it to coolbears.o
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
       -o coolbears-driver coolbears-driver.c \
       coolbears.o
./coolbears-driver
```

```
# this just runs a command but ensures it is built
# first directive runs by default
# usually you should but put the top level build directive here
run: coolbears-driver
        ./coolbears-driver # just a shell command

# build an object file
coolbears.o: coolbears.c
        gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
        -c coolbears.c

# build an executable
coolbears-driver: coolbears-driver.c coolbears.o
        gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
        -o coolbears-driver coolbears-driver.c \
        coolbears.o

# clean is idiomatic for remove object files and executables
clean:
        rm coolbears.o coolbears-driver || echo nothing to delete
```

let's run it. Normally make just runs Makefile. But if you have your own makefiles you should use the -f option with make.

```
make -f Makefile.coolbears clean
make -f Makefile.coolbears coolbears.o
make -f Makefile.coolbears coolbears-driver
make -f Makefile.coolbears run
```

```
rm coolbears.o coolbears-driver || echo nothing to delete
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
        -c coolbears.c
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
-o coolbears-driver coolbears-driver.c \
coolbears.o
./coolbears-driver # just a shell command
The coolest bear is Kevin
```

Or we could just do this:

```
# I am making clean just to clear out the executables and object files
make -f Makefile.coolbears clean
make -f Makefile.coolbears run

rm coolbears.o coolbears-driver || echo nothing to delete
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
        -c coolbears.c
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
-o coolbears-driver coolbears-driver.c \
coolbears.o
./coolbears-driver # just a shell command
The coolest bear is Kevin
```

OR we could do this!

```
# I am making clean just to clear out the executables and object files
make -f Makefile.coolbears clean
make -f Makefile.coolbears

rm coolbears.o coolbears-driver || echo nothing to delete
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
        -c coolbears.c
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
-o coolbears-driver coolbears-driver.c \
coolbears.o
./coolbears-driver # just a shell command
The coolest bear is Kevin
```

Ha it does the same thing!
What if I run make again?

```
# I am making clean just to clear out the executables and object files
make -f Makefile.coolbears

./coolbears-driver # just a shell command
The coolest bear is Kevin
```

It just uses the old object files.

```
# if I remove the executable it'll rebuild only the executable
rm coolbears-driver
make -f Makefile.coolbears
```

```
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
-o coolbears-driver coolbears-driver.c \
coolbears.o
./coolbears-driver # just a shell command
The coolest bear is Kevin

# if I remove the object files it'll build the whole thing
rm *.o
make -f Makefile.coolbears

gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
        -c coolbears.c
gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
-o coolbears-driver coolbears-driver.c \
coolbears.o
./coolbears-driver # just a shell command
The coolest bear is Kevin
```

Personally I would've preferred if assignments were done this way.

1.5.2 DRY Makefile

DRY means DON'T REPEAT YOURSELF.

Let's make a makefile that is easier to use and less prone to errors by repeating text.

Instead of this:

```
# this just runs a command but ensures it is built
run: coolbears-driver
        ./coolbears-driver # just a shell command

# build an object file
coolbears.o: coolbears.c
        gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
        -c coolbears.c

# build an executable
coolbears-driver: coolbears-driver.c coolbears.o
        gcc  -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
        -o coolbears-driver coolbears-driver.c \
        coolbears.o
```

```
# clean is idiomatic for remove object files and executables
clean:
        rm coolbears.o coolbears-driver || echo nothing to delete
```

We're going to automate our Makefile a little more with variables.

You can make a variable in a makefile by going VARNAME=some string of stuff LISTNAME=item1 item2 item3 item4 SCALARNAME="SCALAR VALUE"

```
# common arguments for GCC
CFLAGS= -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3
# Common compiler
CC=gcc
BUILDABLES=coolbears.o coolbears-driver

# this just runs a command but ensures it is built
run: coolbears-driver
        ./coolbears-driver # just a shell command

coolbears.o: coolbears.c
        $(CC) $(CFLAGS) \
        -c coolbears.c

# build an executable
coolbears-driver: coolbears-driver.c coolbears.o
        $(CC) $(CFLAGS) \
        -o coolbears-driver coolbears-driver.c \
        coolbears.o

# clean is idiomatic for remove object files and executables
clean:
        rm $(BUILDABLES) || echo nothing to delete
```

let's run it. Normally make just runs Makefile. But if you have your own makefiles you should use the -f option with make.

```
make -f Makefile.coolbears.dry run
```

```
gcc -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
        -c coolbears.c
gcc -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 \
-o coolbears-driver coolbears-driver.c \
coolbears.o
./coolbears-driver # just a shell command
The coolest bear is Kevin
```

1.5.3 Idiomatic GCC Makefile

Make knows a lot about C. Make comes with default rules that will call your compiler for you as long as CFLAGS and CC are properly set!

This means it knows how to make an executable.

It knows how to make an object file. It just needs to know the dependencies.

You can reuse this makefile as well!

```
# common arguments for GCC
CFLAGS= -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3
# Do you need your math lib? Put it here this is the linking libraries
# variable
LDLIBS=-lm
# Common compiler
CC=gcc
OBJECTS=coolbears.o coolbears-driver.o
EXEC=coolbears-driver
BUILDABLES=$(OBJECTS) $(EXEC)

# this just runs a command but ensures it is built
run: $(EXEC)
        ./$(EXEC) # just a shell command

# We don't even need to specify how to make coolbears.o
# try commenting and uncommenting this line
# coolbears.o: coolbears.c

# build an executable
# coolbears-driver: coolbears-driver.c coolbears.o

# # this would work too
# coolbears-driver: coolbears-driver.o coolbears.o

$(EXEC): $(OBJECTS)


# clean is idiomatic for remove object files and executables
clean:
        rm $(BUILDABLES) || echo nothing to delete
```

let's run it. Normally make just runs Makefile. But if you have your own makefiles you should use the -f option with make.

```
make -f Makefile.coolbears.idiomatic clean
make -f Makefile.coolbears.idiomatic run

rm coolbears.o coolbears-driver.o coolbears-driver || echo nothing to delete
gcc -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3   -c -o coolbears-driver.o coolbear
gcc -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3   -c -o coolbears.o coolbears.c
gcc   coolbears-driver.o coolbears.o  -lm -o coolbears-driver
./coolbears-driver # just a shell command
The coolest bear is Kevin
```

1.5.4 Special Macro Vars

- $@ the target file name
- $< the first dependency
- $? new dependencies that have changed
- $ˆ all dependencies
- $* target suffix

```
OBJECTS=example.txt 1.txt 2.txt 3.txt

run: example.txt
        echo $@ $< $*
        cat example.txt

example.txt: 1.txt 2.txt 3.txt
        echo First Dependency $<
        echo Target $@
        echo New Deps $?
        echo All Deps $^
        echo target suffix $*
        cat $^ > example.txt

1.txt:
        echo $@ > $@
2.txt:
        echo $@ > $@
3.txt:
        echo $@ > $@

clean:
        rm $(OBJECTS) || echo all good
```

```
make -f Makefile.macros clean
make -f Makefile.macros
```

```
rm example.txt 1.txt 2.txt 3.txt || echo all good
echo 1.txt > 1.txt
echo 2.txt > 2.txt
echo 3.txt > 3.txt
echo First Dependency 1.txt
First Dependency 1.txt
echo Target example.txt
Target example.txt
echo New Deps 1.txt 2.txt 3.txt
New Deps 1.txt 2.txt 3.txt
echo All Deps 1.txt 2.txt 3.txt
All Deps 1.txt 2.txt 3.txt
echo target suffix
target suffix
cat 1.txt 2.txt 3.txt > example.txt
echo run example.txt
run example.txt
cat example.txt
1.txt
2.txt
3.txt
```

1. Implicit Rules

   Here's an example of implicit rules. Of how we convert 1 file to another implicitly much like how Make hands C compilation.

   ```
   ############################### Implicit rules ###############################

   # Convert a .tex file to a .pdf
   %.pdf: %.tex $(ALLDEPS)
           latexmk -pdf $(LATEXMK_OPTS) $*

   # Convert SVGs to PDFs
   # Requires Inkscape
   %.pdf: %.svg
           inkscape -b white -t -T --export-ignore-filters --export-pdfs=$@ $<

   # Convert EPSs to PDFs
   # epstopdf(1) is often bundled with TeX distributions
   %.pdf: %.eps
           epstopdf $<

   # Automatically crops the margins of a PDF.
   %-crop.pdf: %.pdf
           pdfcrop $<
   ```

1.5.5 Makefile help

The GNU Manual is pretty good
`https://www.gnu.org/software/make/manual/html_node/Introduction.html`