CMPUT201W20B2 Week 2

Abram Hindle

March 10, 2020

Contents

1 Week2

1.1 Copyright Statement

1.1.1 Hazel Code is licensed under AGPL3.0+

1.1.2 Apache 2 Preamble

1.2 Init ORG-MODE

;; I need this for org-mode to work well

(require 'ob-sh) ;(require 'ob-shell) (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)

1.2.1 Org export

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

1.3 Org Template

Copy and paste this to demo C

```
#include <stdio.h>

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

1.4 Remember how to compile?

gcc -std=c99 -Wall -pedantic -Werror -o programname programname.c

1.5 Unary versus Binary

```
#include <stdio.h>

int main() {
    int an_int = -5; // unary
    int another_int = an_int + an_int; //binary
    printf("an_int=\t%d\n", an_int);
    printf("another_int=\t%d\n", another_int);
    ++an_int;
    another_int = -(an_int) + +(an_int); //binary and unary
    printf("another_int=\t%d\n", another_int);
    another_int = -(an_int) + -(an_int); //binary and unary
    printf("another_int=\t%d\n", another_int);
    return 0;
}
```

```
an_int= -5
another_int= -10
another_int= 0
another_int= 8
```

1.6 Arithmetic

Arithmetic operators have precedence.

```
#include <stdio.h>

int main() {
    int an_int = 5;
    int another_int = an_int + an_int;
    printf("another_int=%d\n", another_int);

    another_int = an_int * an_int;
    printf("(an_int*an_int) another_int=%d\n", another_int);
    int order1 = an_int * an_int + an_int * an_int / an_int;
    int order2 = (an_int * an_int) + ((an_int * an_int) / an_int);
    int order3 = an_int * (an_int + (an_int * (an_int / (an_int))));
    if ( !((order1) == order2) ) {
        printf("order1 != order2\n");
    } else {
        printf("order1 == order2\n");
    }
    if (order1 == order3) {
        printf("order1 == order3\n");
    } else {
        printf("order1 != order3\n");
    }

}
```

```
another_int=10
(an_int*an_int) another_int=25
order1 == order2
order1 != order3
```

1.7 Assignments in Expressions [Hazel Example]

This is a popular feature of C that causes a lot of bugs and lot of confusion with C.

```
#include <stdio.h>

int main() {
    int an_int = 5;
    printf("an_int=%d\n", an_int);
    // We can use an assignment as an expression!
    printf("(an_int = 2)=%d\n", an_int = 2);
    printf("(an_int = 5) > 5: ");
    if ((an_int = 5) > 5) {
        printf("true\n");
    } else {
        printf("false\n");
    }
    printf("(an_int *= 3) > 5: ");
    if ((an_int *=  3) > 5) {
        printf("true\n");
    } else {
        printf("false\n");
    }
    printf("an_int is now = %d\n", an_int);
    return 0;
}
```

```
an_int=5
(an_int = 2)=2
(an_int = 5) > 5: false
(an_int *= 3) > 5: true
an_int is now = 15
```

1.8 Order of operations from Hazel

Order of operations for l-values is from right to left

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

```
int main() {
    int one = 1;
    int two = 2;
    int three = one = two;     // does three == one or does three == two
    printf("one=%d\n", one);
    printf("two=%d\n", two);
    printf("three=%d\n", three);
    printf("(one++)=%d (one*=2)=%d\n", one++, one *= 2);
    one = 2;
    one *= 2;
    //one++
    int tmp = one;
    one += 1;
    printf("(one++)=%d (one*=2)=%d\n", tmp, one);

    return 0;
}
```

```
one=2
two=2
three=2
(one++)=4 (one*=2)=5
(one++)=4 (one*=2)=5
```

1.9 L-value from Hazel

```
#include <stdio.h>

#define N 5

int main() {
    int an_int = 5;
    printf("an_int = 5\n");
    printf("an_int=%d\n", an_int);
    an_int = 2;
    printf("an_int = 2\n");
    printf("an_int=%d\n", an_int);
    // an_int is an "lvalue"
    // aka. left value
    // anything that can appear on the left of an assignment
```

```
    // we can also use ++ and -- with lvalues
    an_int += 20;
    printf("an_int += 20\n");
    printf("an_int=%d\n", an_int);
    printf("an_int++=%d\n", an_int++);
    printf("an_int=%d\n", an_int);
    printf("++an_int=%d\n", ++an_int);
    printf("an_int=%d\n", an_int);
    an_int *= 10;
    printf("an_int *= 10\n");
    printf("an_int=%d\n", an_int);
    printf("an_int--=%d\n", an_int--);
    printf("an_int=%d\n", an_int);
    printf("--an_int=%d\n", --an_int);
    printf("an_int=%d\n", an_int);
    // None of the following work, because they are NOT lvalues!
    // 5 = 2;
    // 5 *= 10;
    // 5++;
    // N = 2;
    // N *= 10;
    // N++;
    return 0;
}

an_int = 5
an_int=5
an_int = 2
an_int=2
an_int += 20
an_int=22
an_int++=22
an_int=23
++an_int=24
an_int=24
an_int *= 10
an_int=240
an_int--=240
an_int=239
--an_int=238
```

```
an_int=238
```

1.10 Boolean Values

C has an idea of truthy values and false values.

A 0, null, or 0.0f is a false value.

Anything else is a true value. This means that all arrays and strings are true. All characters except \0 are true. All floating point values that are not 0.0f or -0.0f are true.

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

int main() {
    char chars[1024];
    char emptystring[1024] = "";
    if ( 0.0f ) {
        printf("0.0f is true!\n");
    } else {
        printf("0.0f is not true!\n");
    }
    if ( -0.0f ) {
        printf("-0.0f is true!\n");
    } else {
        printf("-0.0f is not true!\n");
    }
    if ( emptystring ) {
        printf("emptystring is true!\n");
    } else {
        printf("emptystring is not true!\n");
    }
    if ( chars ) {
        printf("character array is true!\n");
    } else {
        printf("character array is not true!\n");
    }
    if ( 0 ) {
        printf("0 is true!\n");
    } else {
```

```
        printf("0 is not true!\n");
    }
    if ( emptystring[0] ) {
        printf("emptystring[0] is true!\n");
    } else {
        printf("emptystring[0] is not true!\n");
    }
    char * nostringatall = NULL;
    if ( nostringatall ) {
        printf("nostringatall is true!\n");
    } else {
        printf("nostringatall is not true!\n");
    }

    return 0;
}
```

```
0.0f is not true!
-0.0f is not true!
emptystring is true!
character array is true!
0 is not true!
emptystring[0] is not true!
nostringatall is not true!
```

1.11 Boolean Values from Hazel

```
./bool.c
```

```
gcc -v -g -O0 -std=c99 -Wall -pedantic -o bool bool.c
```

```
echo 2 | ./bool
```

enter a number from 0-3: a is two ($a_{number} < 2$)=0 ($a_{number} == 2$)=1 true=1 false=0 lt_{two}=0 (lt_{two}): false lt_{twoint}=0 (lt_{twoint}): false (1 = true): 1 (2 = true): 0 4 < a_{number} < 10: true ($a_{number} < 2$? 111 : 222)=222

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

```
int main() {
    int a_number = -1;
    printf("enter a number from 0-3: ");
    int scanf_result = scanf("%d", &a_number);
    if (scanf_result != 1) {
        printf("Error: didn't read a number!\n");
        abort();
    }

    if (a_number < 2) {
        printf("a less than two\n");
    } else if (a_number == 2) {
        printf("a is two\n");
    } else {
        printf("a is greater than two\n");
    }

    printf("(a_number < 2)=%d\n", a_number < 2);

    printf("(a_number == 2)=%d\n", a_number == 2);

    // using true and false by name: be sure to include <stdbool.h>
    printf("true=%d\n", true);
    printf("false=%d\n", false);

    bool lt_two = a_number < 2;
    printf("lt_two=%d\n", lt_two);

    printf("(lt_two): ");
    if (lt_two) {
        printf("true\n");
    } else {
        printf("false\n");
    }

    int lt_two_int = a_number < 2;
    printf("lt_two_int=%d\n", lt_two_int);

    printf("(lt_two_int): ");
    if (lt_two_int) {
```

```
        printf("true\n");
    } else {
        printf("false\n");
    }

    // NEVER do == true, because any number that's not 0 is true.
    printf("(1 == true): %d\n", 1 == true);
    printf("(2 == true): %d\n", 2 == true);


    // to fix this 4 < a && a < 10
    printf("4 < a_number < 10: ");
    if ((4 < a_number) && (a_number < 10)) {
        printf("true\n");
    } else {
        printf("false\n");
    }
    int tmp = 0;
    if (a_number < 2) {
       tmp = 111;
    } else {
       tmp = 222;
    }
    tmp = (a_number < 2 ? 111 : 222);
    a_number < 2 ? tmp = 111 : tmp = 222;
    printf("(a_number < 2 ? 111 : 222)=%d\n", tmp);
    printf("(a_number < 2 ? 111 : 222)=%d\n", (a_number < 2 ? 111 : 222));
    (a_number < 2 ? 111 :((a_number < 1)? 000 : 222));
    return 0;
}
```

1.12 Pre and Post Increment

- x++ and ++x both eventually increment x but each does something different.
- ++x increments x and returns x's incremented value
- x++ returns x and then increments x's value after.

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

```
int main() {
    int x = 0;
    printf("x = %d\n", x);
    printf("++x ~ %d\n", ++x);
    printf("x = %d\n", x);
    printf("x++ ~ %d\n", x++);
    printf("x = %d\n", x);
    return 0;
}
```

```
x = 0
++x ~ 1
x = 1
x++ ~ 1
x = 2
```

1.12.1 Code for x++ and ++x

```
#include <stdio.h>

int main() {
    int x = 0;
    printf("x = %d\n", x);
    // preincrement
    // ++x;
    x = x + 1;
    printf("++x ~ %d\n", x);
    printf("x = %d\n", x);

    // postincrement
    // x++;
    int tmp = x;
    x = x + 1;
    // note that our expression has changed from x++ to tmp because we return
    // the prior value of x and it is incremented afterwards
    printf("x++ ~ %d\n", tmp);
    printf("x = %d\n", x);
    return 0;
}
```

```
x = 0
++x ~ 1
x = 1
x++ ~ 1
x = 2

   ./pre-post.c

#include <stdio.h>

int main() {
    int x = 100;
    int y = x++;
    int z = ++x;
    return y;
}

gcc -v -g -O0 -std=c99 -Wall -pedantic -o pre-post pre-post.c
```

1.12.2 Objdump it!

```
gcc -v -g -O0 -std=c99 -Wall -pedantic -o pre-post pre-post.c
objdump -d -S pre-post
```

1.12.3 Objdump Main

#+BEGIN$_{\text{SRC}}$ verbatim 00000000000005fa <main>: #include <stdio.h>
int main() { 5fa: 55 push %rbp # store main on the stack 5fb: 48 89 e5 mov %rsp,%rbp # move the stackpointer to rbp int x = 100; 5fe: c7 45 f4 64 00 00 00 movl $0x64,-0xc(%rbp) # set x on the stack to 100 int y = x++; 605: 8b 45 f4 mov -0xc(%rbp),%eax # copy x to eax 608: 8d 50 01 lea 0x1(%rax),%edx # copy x+1 to edx (GCC why you abuse lea?!) 60b: 89 55 f4 mov %edx,-0xc(%rbp) # copy x+1 back to the stack as x 60e: 89 45 f8 mov %eax,-0x8(%rbp) # store old x from eax into where y is stored int z = ++x; 611: 83 45 f4 01 addl $0x1,-0xc(%rbp) # add 1 to x on the stack 615: 8b 45 f4 mov -0xc(%rbp),%eax # copy x from stack to eax 618: 89 45 fc mov %eax,-0x4(%rbp) # store it into z return y; 61b: 8b 45 f8 mov -0x8(%rbp),%eax # copy y into eax to return } 61e: 5d pop %rbp # restore base pointer 61f: c3 retq # return #+END$_{\text{SRC}}$ verbatim

1.13 Comma Operator from Hazel

```
#include <stdio.h>

/* The comma operator:
 * You should never use it in your own code!
 * It evaluates the expression on the left side of the comma, and discards the result.
 * Then it evaluates the expression on the right side. The value on the right side is t
 */

int main() {
    int an_int;
    int array[2] = { 100, 200 };
    // look at the inconsistency between an expression
    an_int = (1,2,3);
    printf("%d\n", an_int);
    // an a direct assignment (take the left most)
    an_int = 1,2,3;
    printf("%d\n", an_int);
    // first do (1,2) in expression order (take the right)
    // then do 2,3 in assignment order! (take the left)
    an_int = (1,2),3;
    printf("%d\n", an_int);
    printf("Don't rely on the comma operator!");
    return 0;
}

3
1
2
Don't rely on the comma operator!
```

1.14 If statements

if statements are of the form:

```
if ( condition ) {
   code for true case;
}

if ( condition ) {
```

```
   code for true case;
} else {
   code for false case;
}

if ( condition ) {
   code for true case;
} else if ( condition2 ) {
   code for !condition && condition2 case
} else {
   code for false case;
}

&i // where my int is
i & i // i and i
i | i // i or i
i ^ i // i xor i
i && i // i logical and i

#include <stdio.h>

int main(int argc, char**argv) {

   int input = 0;
   int condition =  input > 1;
   int condition2 = input == 1;
   int condition3 = input > 0;

   if ( condition ) {
      printf("if code for true case\n");
   }

   if ( condition ) {
      printf("if-else code for true case\n");
   } else {
      printf("if-else code for false case\n");
   }

   if ( condition ) {
      printf("if-else-if-else code for condition is true case\n");
```

```
   } else if ( condition2 ) {
      printf("if-else-if-else code for !condition && condition2 case\n");
   } else {
      printf("if-else-if-else code for !condition && !condition2 case\n");
   }

   if (condition) {
      printf("parallel-ifs condition!\n");
   }
   if (condition2) {
      printf("parallel-ifs condition2!\n");
   }
   if (condition3) {
      printf("parallel-ifs condition3!\n");
   }

    return 0;
}
```

```
if-else code for false case
if-else-if-else code for !condition && !condition2 case
```

1.15 Blocks! from Hazel

Block are chunks of code that are related or grouped together by the parser and compiler. The code that is executed on the else statement of an if is a block.

Many C statements like if do not need { } braces for blocks

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

int main() {
    int some_num = 3;

    // mildly confusing syntax
    printf("Will print 0 if 0 is true:\n");
    if (0) printf("0\n");
    // printf("the real false 0\n");
```

```
    // more confusing syntax
    printf("Will print 1 if 1 is true:\n");
    if (1)
        printf("1\n");
    printf("the real true 1\n");
    // really confusing syntax
    if (2) printf("2 is true\n");
    else printf("2 is false\n");

    printf("------------------------------\n");

    // good syntax! Do this! Use braces!
    printf("outside of if some_num=%d\n", some_num);
    if (1) {
        int some_num = 2;
        printf("inside of if some_num=%d\n", some_num);
    }
    printf("outside of if some_num=%d\n", some_num);

    return 0;
}
```

```
Will print 0 if 0 is true:
Will print 1 if 1 is true:
1
the real true 1
2 is true
------------------------------
outside of if some_num=3
inside of if some_num=2
outside of if some_num=3
```

1.16 Ternary Statements

An if statement that is an expression!

(condition) ? (true expression) : (false expression)

Only use 1 ternary at a time. More is confusing.

/ don't do this / (condition $>$ 0) * (true expression) + (condition $==$ 0) * (false expression)

int functionName(int xx) { int x = (xx == 0)?16:xx; }

```
#include <stdio.h>

int main(int argc, char**argv) {
    int a_number = 0;
    int tmp = (a_number < 2 ? 111 : 222);
    // don't do this
    int res = a_number < 2 ? (tmp = 111) : (tmp = 222);
    printf("res = a_number < 2 ? tmp = 111 : tmp = 222? res = %d\n", res);
    printf("(a_number < 2 ? 111 : 222)=%d\n", tmp);
    printf("(a_number < 2 ? 111 : 222)=%d\n", (a_number < 2 ? 111 : 222));
    printf("An expression? %d\n", (a_number < 2 ? 111 :((a_number < 1)? 000 : 222)));
    return 0;
}
```

```
res = a_number < 2 ? tmp = 111 : tmp = 222? res = 111
(a_number < 2 ? 111 : 222)=111
(a_number < 2 ? 111 : 222)=111
An expression? 111
```

1.17 Switch Statements

Switch statement operators on values and matches values directly. If you use it with strings it will surprise you and not work as expected. Use it on characters, bools, ints. Try not to use it on floats.

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

int main() {
    char direction = 'u';
    switch(direction) {
        case 'u':
        case 'U':
            printf("Up\n");
            break;
        case 'D':
            printf("Down\n");
            break;
```

```
        case 'L':
            printf("Left\n"); // bug here
        case 'R':
            printf("Right\n");
            break;
        default:
            printf("Direction is invalid");
    }
    return 0;
}
```

Up

```
// Copyright (c) 2020 Hazel Campbell
// Licensed under AGPL3.0+
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    char depth = 2;
    switch(depth) {
        case 0:
            printf("0\n");
        case 1:
            printf("1111\n");
        case 2:
            printf("22222222\n");
        case 3:
            printf("333333333333\n");
        case 4:
            printf("4444444444444444\n");
            break;
        default:
            printf("Too big or too small");
    }
    return 0;
}
```

22222222

```
333333333333
4444444444444444
```

1.18 Switch Statements [code from Hazel]

./switch.c

```
gcc -v -g -O0 -std=c99 -Wall -pedantic -o switch switch.c

echo input  1
echo 1 | ./switch
echo input  2
echo 2 | ./switch
echo input  3
echo 3 | ./switch
echo input  4
echo 4 | ./switch
echo input  5
echo 5 | ./switch

input 1
enter number number from 0-3: one
input 2
enter number number from 0-3: two
two or three
input 3
enter number number from 0-3: two or three
input 4
enter number number from 0-3: not a number from 0-3
input 5
enter number number from 0-3: not a number from 0-3

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

int main() {
    int number = -1;
    printf("enter number number from 0-3: ");
    int scanned = scanf("%d", &number);
    if (scanned != 1) {
```

```
        printf("Eh?\n");
        abort();
    }
    switch(number) {
        case 0:
            printf("zero\n");
            break;
        case 1:
            printf("one\n");
            break;
        case 2:
            printf("two\n");
        case 3:
            printf("two or three\n");
            break;
        default:
            printf("not a number from 0-3\n");
    }
    return 0;
}
```

1.19 While loops

```
#include <stdio.h>

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

1.20 For loops

```
for ( init_condition; looping_condition ; runs_each_time )

// Loop from 0 to 10
for ( int i = 0 ; i <= 10 ; i++ )
for ( int i = 0 ; i <= 10 ; i+=1 )
for ( int i = 0 ; i <= 10 ; ++i )


#include <stdio.h>
```

```
int main() {
   //int i = 0;
   printf("i++\n");
   for ( int i = 0 ; i < 10 ; i++ ) {
       printf("%d ", i);
   }
   // printf("i %d", i); // i is undeclared in this scope!
   printf("\ni+=1\n");
   for ( int i = 0 ; i < 10 ; i+=1 )  {
       printf("%d ", i);
   }
   printf("\n++i\n");
   for ( int i = 0 ; i < 10 ; ++i )  {
       printf("%d ", i);
   }
   printf("\ni+=3\n");
   int i = 0;
   for ( i = 0 ; i < 10 ; i+=3 )  {
       printf("%d ", i);
   }
   printf("\ni left behind %d\n", i);
   return 0;
}
```

```
i++
0 1 2 3 4 5 6 7 8 9
i+=1
0 1 2 3 4 5 6 7 8 9
++i
0 1 2 3 4 5 6 7 8 9
i+=3
0 3 6 9
i left behind 12
```

1.20.1 Idiomatic For Loops

Here's how you write good for loops in C99

```
// Copyright (c) 2020 Hazel Campbell
// Licensed under the AGPL3.0+
```

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

int main() {
    int count_to = 4;
    // Counting from 0 to count_to-1 (3)
    for (int counter = 0; counter < count_to; counter++) {
        printf("%d ", counter);
    }
    printf("\n");
    // Counting from 1 to count_to (4)
    for (int counter = 1; counter <= count_to; counter++) {
        printf("%d ", counter);
    }
    printf("\n");
    // Counting from count_to-1 to 0
    for (int counter = count_to-1; counter >= 0; counter--) {
        printf("%d ", counter);
    }
    printf("\n");
    // Counting from count_to to 1
    for (int counter = count_to; counter >= 1; counter--) {
        printf("%d ", counter);
    }
    printf("\n");
    return 0;
}
```

```
0 1 2 3
1 2 3 4
3 2 1 0
4 3 2 1
```

1.20.2 Hazel's Gallery of Misfit For Loops

```c
// Copyright (c) 2020 Hazel Campbell
// Licensed under AGPL3.0+
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>

#define COUNT_TO 5

int main() {
    int counter;
    // clear idiomatic for loop!
    for (counter = 0; counter < COUNT_TO; counter++) {
        printf("%d\n", counter);
    }
    printf("---\n");
    counter = 0;
    // while loop that does the same thing :/
    while (counter < COUNT_TO) {
        printf("%d\n", counter);
        counter++;
    }
    printf("---\n");
    printf("%d\n", counter);
    // for (
    //  something that runs once ;
    //  counter conditional that's checked
    //      each time;
    //  something that happens each time ;
    // )
    printf("---\n");
    counter = 0;
    for (; counter < COUNT_TO; counter++) {
        printf("%d\n", counter);
    }
    printf("---\n");
    counter = 0;
    for (; counter < COUNT_TO; counter++) {
        printf("%d\n", counter);
        counter = COUNT_TO;
    }
    printf("---\n");
    printf("%d\n", counter);
    printf("---\n");
    counter = 0;
```

```
    for (; counter < COUNT_TO;) {
        printf("%d\n", counter);
        counter = COUNT_TO;
    }
    for (;;) {
       // infinite loop here
       break;
    }
    printf("---\n");
    printf("%d\n", counter);
    return 0;
}
```

```
0
1
2
3
4
---
0
1
2
3
4
---
5
---
0
1
2
3
4
---
0
---
6
---
0
---
5
```

1.21 Break and For

The break statement allows you to quit the immediate for loop or while loop you are in.

```
// Copyright (c) 2020 Hazel Campbell
// Licensed under AGPL3.0+

#include <stdio.h>

int main() {
    int count_to = 4;
    for (int i = 0; i < count_to; i++) {
        for (int j = 0; j < count_to; j++) {
            if (i+j == count_to) {
                break;
            }
            printf("%d ", i+j);
        }
        printf("\n");
    }
}
```

```
0 1 2 3
1 2 3
2 3
3
```

1.21.1 Continue

Continue is like break except it skips the loop body to the end and restarts.

```
// Copyright (c) 2020 Hazel Campbell
// Licensed under AGPL3.0+

#include <stdio.h>

int main() {
    int count_to = 4;
    for (int i = 0; i < count_to; i++) {
        for (int j = 0; j < count_to; j++) {
```

```
            if (i+j == count_to) {
                continue;
            }
            printf("%d ", i+j);
        }
        printf("\n");
    }
}
```

```
0 1 2 3
1 2 3
2 3 5
3 5 6
```

1.22 GOTO

GOTO is basically our jmp opcode from assembler except we can only go to labels.

labels in code are an identifier followed by a colon:

If you have -g flags in gcc the label will be in executable, otherwise it will just be an offset that you jump to.

GOTOs can only be in the same function in C.

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

int main() {
    goto super;
    ala:
    printf("Aladocious\n");
    goto end;
    expi:
    printf("Expi\n");
    goto ala;
    fragi:
    printf("Fragilistic\n");
    goto expi;
    cali:
    printf("Cali\n");
```

```
    goto fragi;
    super:
    printf("Super\n");
    goto cali;
    end:
    printf("Done! Don't use GOTOs!");
}
```

```
Super
Cali
Fragilistic
Expi
Aladocious
Done! Don't use GOTOs!
```

```
// Copyright (c) 2020 Hazel Campbell
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define STOP_BEFORE 7

int main() {
    for (int counter = 0; counter < STOP_BEFORE; counter++) {
        switch (counter) {
            case 0:
                printf("zero ");
                break;
            case 1:
                printf("one ");
                break;
            case 2:
                printf("two ");
                break;
            case 3:
                printf("three ");
                break;
            default:
                printf("error!\n");
```

```
                goto done;
                break;
        }
    }
    done:
    printf("\n");
    return 0;
}
```

```
zero one two three error!
```

1. Exercise

   for (int counter = 0; counter < 60 ; counter += 3) { for (int counter = 1; counter < 60 ; counter *= 3) { for (int i = 64; i > 0; i /= 2) { printf("%d",i); }

   ```
   // Copyright (c) 2020 Hazel Campbell
   #include <stdio.h>
   #include <stdlib.h>
   #include <string.h>

   #define STOP_BEFORE 7

   int main() {
        for ( int i = 64;  i > 0; i /= 2)  {
           printf("%d\n",i);
        }
        return 0;
   }
   ```

   ```
   64
   32
   16
   8
   4
   2
   1
   ```

1.23 Types!

1.23.1 int!

In C ints are often 32-bit integers. They can have a sign.

```
#include <stdio.h>
#include <limits.h>
int main() {
    int an_int = 6;
    printf("size_of(an_int) == %ld\n", sizeof(an_int));
    int max_int = INT_MAX;
    printf("max int == %11d\n", max_int);
    int min_int = INT_MIN;
    printf("min int == %11d\n", min_int);
    printf("an_int  == %11d \t== 0x%08x\n", an_int,  an_int);
    printf("min_int == %11d \t== 0x%08x\n", min_int, min_int);
    printf("max_int == %11d \t== 0x%08x\n", max_int, max_int);
    printf("     -1 == %11d \t== 0x%08x\n", -1,-1);
    printf("      1 == %11d \t== 0x%08x\n", 1,1);
    printf("      0 == %11d \t== 0x%08x\n", 0,0);

    return 0;
}
```

```
size_of(an_int) == 4
max int ==  2147483647
min int == -2147483648
an_int  ==           6  == 0x00000006
min_int == -2147483648  == 0x80000000
max_int ==  2147483647  == 0x7fffffff
     -1 ==          -1  == 0xffffffff
      1 ==           1  == 0x00000001
      0 ==           0  == 0x00000000
```

1.23.2 unsigned ints!

You can only non-negative integers if you want

```
#include <stdio.h>
#include <limits.h>
int main() {
```

```
    unsigned int an_int = 6;
    printf("size_of(an_int) == %ld\n", sizeof(an_int));
    unsigned int max_int = UINT_MAX;
    printf("max int == %11u\n", max_int);
    unsigned int min_int = 0;
    printf("min int == %11u\n", min_int);
    printf("an_int  == %11u \t== 0x%08x\n", an_int,  an_int);
    printf("min_int == %11u \t== 0x%08x\n", min_int, min_int);
    printf("max_int == %11u \t== 0x%08x\n", max_int, max_int);
    printf("     -1 == %11u \t== 0x%08x\n", -1,-1);
    printf("      1 == %11u \t== 0x%08x\n", 1,1);
    printf("      0 == %11u \t== 0x%08x\n", 0,0);
    return 0;
}

size_of(an_int) == 4
max int ==  4294967295
min int ==           0
an_int  ==           6  == 0x00000006
min_int ==           0  == 0x00000000
max_int ==  4294967295  == 0xffffffff
     -1 ==  4294967295  == 0xffffffff
      1 ==           1  == 0x00000001
      0 ==           0  == 0x00000000
```