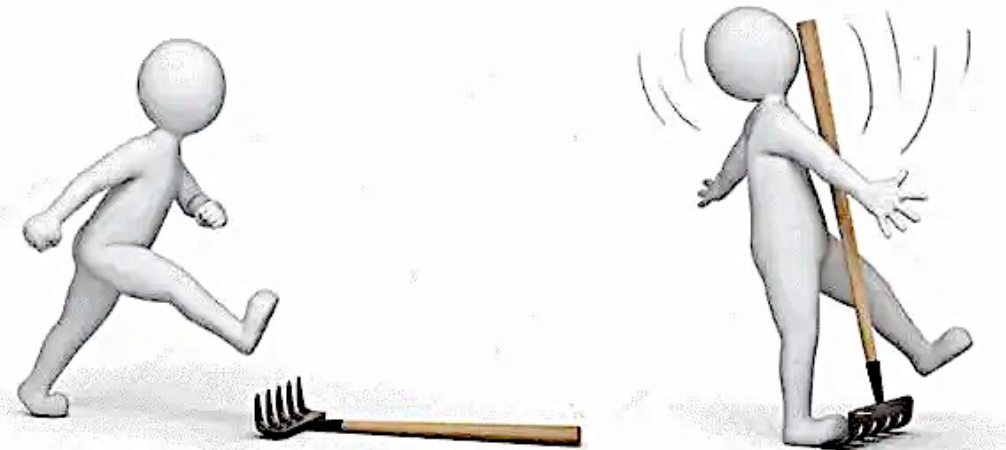
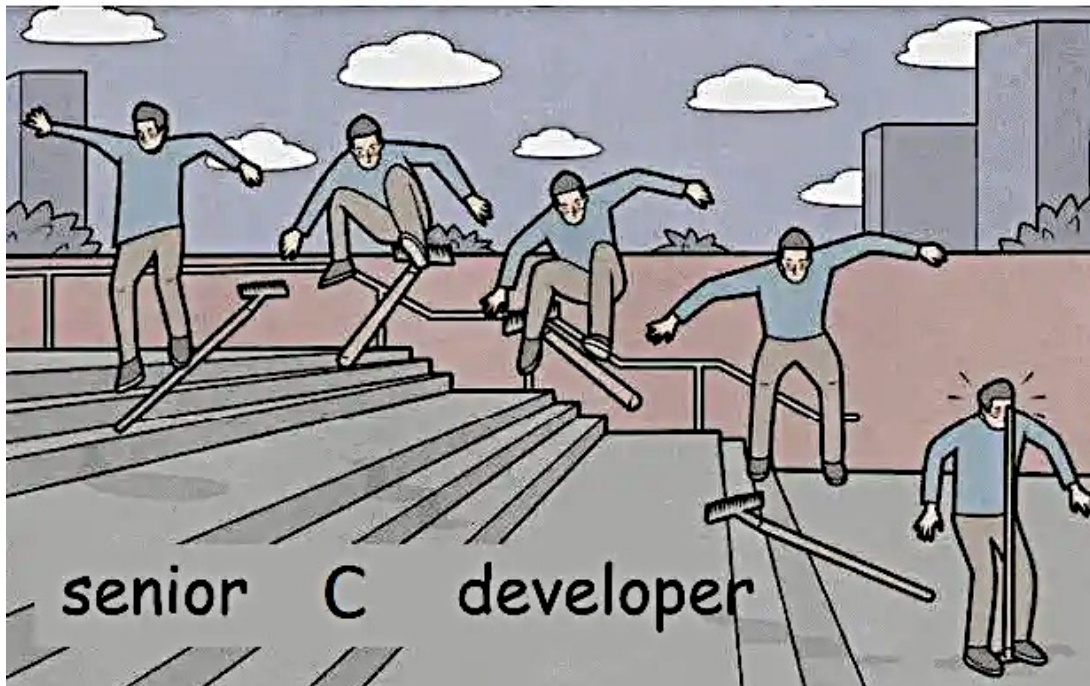


Lecture 8: C part I

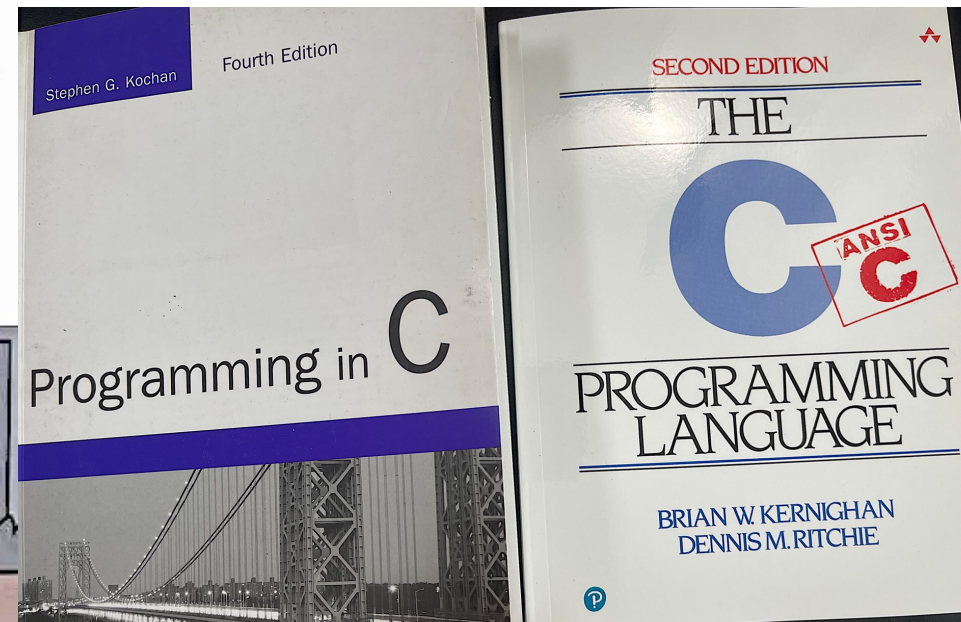


junior C developer



senior C developer

Suggested Literature



Suggested Literature

Both books are available at
<https://learning.oreilly.com>

C is a statically typed compiled language

- To run C/C++/Fortran you need a compiler. Example: the GNU compiler

```
whovian@phy504:~$ gcc --version
gcc (Ubuntu 10.3.0-1ubuntu1~20.04) 10.3.0
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

C compiler

```
whovian@phy504:~$ g++ --version
g++ (Ubuntu 10.3.0-1ubuntu1~20.04) 10.3.0
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

C++ compiler

```
whovian@phy504:~$ gfortran --version
GNU Fortran (Ubuntu 10.3.0-1ubuntu1~20.04) 10.3.0
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Fortran compiler

C is a statically typed compiled language

- Compiler will transform C/C++/Fortran into Assembly language.
- Once a program is compiled, its source code is not useful for running the code. For interpreted programs, the source code is needed to run the program every time (like in bash)
- There are ways to create an interpreter for C/C++/Fortran. Example: Cling is a C++ interpreter. However, it will never feel as fluid/natural as Python, Bash...
- After compilation, the compiler also needs to link the files into a program
- Important: during compilation, different files are not aware of each other (this will be relevant in the future)

Hello World in C

Hello World has many different features compared to bash

```
GNU nano 4.8      helloworld.c
#include <stdio.h>      Standard Library

int main (void)      All C programs need a main
{                    function
    printf ("Physics 504 is fun.\n");

    return 0;      Return value can be used by
}                  bash to check if the program
                  run successfully

C requires ;
at the end of every line
```

Hello World in C

Hello World has many different features compared to **bash**

```
GNU nano 4.8 helloworld.c
#include <stdio.h>

int main (void)
{
    printf ("Physics 504 is fun.\n");
    return 0;
}
```

Arguments of main allow command line arguments to be passed to C

printf here is similar in many ways to Bash

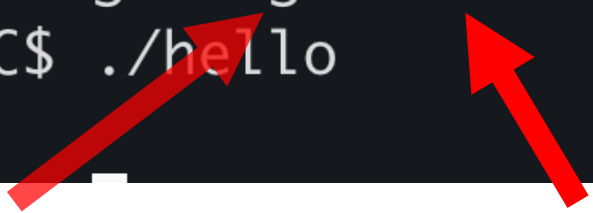
Hello World in C

When you have only 1 file: one command to compile and link the code (we will learn about **Makefiles** later for $N > 1$ files)

```
whovian@phy504:~/host/C$ nano helloworld.c
whovian@phy504:~/host/C$ gcc helloworld.c -o hello
whovian@phy504:~/host/C$ ./hello
Physics 504 is fun.
```

The compiler accepts many options; it can perform conservative or aggressive optimizations, display warnings, or save extra information that is useful for debuggers

```
whovian@phy504:~/host/C$ gcc -g3 -O0 helloworld.c -o hello
whovian@phy504:~/host/C$ ./hello
Physics 504 is fun.
```



provides information useful
for debuggers (e.g., GDB)

disable float-point
optimizations (to enable **O1-3**)

Comments and variables in C

Two commands for comments in C : `"//"` and `"/* ... */"`

```
GNU nano 4.8                                sumint.c
#include <stdio.h>
/*
Multi line observations
Line 2!!
*/
int main (void)
{
    int value1, value2;

    value1 = 50;
    value2 = 25;
    int sum = value1 + value2; // you can declare inline

    printf ("The sum of %i and %i is %i\n", value1, value2, sum);

    return 0;
}
```

whovian@phy504:~/host/C\$ gcc sumint.c -o sum

whovian@phy504:~/host/C\$./sum

The sum of 50 and 25 is 75

**Comments
(not compiled)**



int / float / double types in C

GNU nano 4.8

dtypes.c

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

```
int main (void)
{
```

```
    int        integerVar = 100;
    float       floatingVar = 331.79;
    double      doubleVar = 8.44e+11;
    char        charVar = 'W';
    _Bool       boolVar = 0;
```

```
    printf ("integerVar = %i\n", integerVar);
    printf ("floatingVar = %f\n", floatingVar);
    printf ("doubleVar = %e\n", doubleVar);
    printf ("doubleVar = %g\n", doubleVar);
    printf ("charVar = %c\n", charVar);
    printf ("boolVar = %i\n", boolVar);
```

```
    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc dtypes.c -o dtypes
```

```
whovian@phy504:~/host/C$ ./dtypes
```

```
integerVar = 100
```

```
floatingVar = 331.790009
```

```
doubleVar = 8.440000e+11
```

```
doubleVar = 8.44e+11
```

```
charVar = W
```

```
boolVar = 0
```

[Interesting discussion
about Bool](#)

We have seen **printf** in Bash. Note the differences (e.g., the parentheses "(...)" and the commas ",")

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

```
int main (void)
```

```
{
```

```
    int a = 1;
```

```
    int b = 2;
```

```
    int c = a/b;
```

```
    double d = a/b;
```

```
    double e = a / (double) b;
```

```
    double f = 1/2;
```

```
    double g = 1./2.;
```

```
    printf ("c = %i\n", c);
```

```
    printf ("d = %f\n", d);
```

```
    printf ("e = %f\n", e);
```

```
    printf ("f = %f\n", f);
```

```
    printf ("g = %f\n", g);
```

```
    return 0;
```

```
}
```

Type casting



Integer variables in C

Be careful with integer division

```
whovian@phy504:~/host/C$ gcc intdiv.c -o idiv
```

```
whovian@phy504:~/host/C$ ./idiv
```

```
c = 0
```

```
d = 0.000000
```

```
e = 0.500000
```

```
f = 0.000000
```

```
g = 0.500000
```

```

GNU nano 4.8 floatprec.c Mo
#include <stdio.h>

int main (void)
{
    // float has 10^-7 prec. max
    float a = 1.0012033;
    float b = 1.0012034;
    float c = b - a;

    printf ("c = %g\n", c);

    double f = 1.0012033;
    double g = 1.0012034;
    double h = g - f;

    printf ("h = %g\n", h);

    return 0;
}

```

“Problem” in
EVERY language

float/double
variables in C

[Good discussion on StackOverflow](#) on
float vs double vs
long double

[What Every Computer Scientist Should Know About Floating-Point Arithmetic](#)

```

whovian@phy504:~/host/C$ nano floatprec.c
whovian@phy504:~/host/C$ gcc floatprec.c -o float
whovian@phy504:~/host/C$ ./float
c = 1.19209e-07
h = 1e-07
whovian@phy504:~/host/C$

```

wrong result

Be careful: casting is not rounding

GNU nano 4.8 castvsround.c

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

```
int main (void)
```

```
{
    double a = 2.9;
    int b = (int) a;
    int c = round(a);
```

```
    printf ("b = %i\n", b);
    printf ("c = %i\n", c);
```

```
    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc castvsround.c -o castvsround
/usr/bin/ld: /tmp/cc8BHsBG.o: in function `main':
castvsround.c:(.text+0x2f): undefined reference to `round'
collect2: error: ld returned 1 exit status
```

```
whovian@phy504:~/host/C$ g++ castvsround.c -o castvsround
```

We must link the math
library manually
(historical reasons)
(btw: not in C++)

```
whovian@phy504:~/host/C$ gcc castvsround.c -o castvsround -lm
whovian@phy504:~/host/C$ ./castvsround
```

```
b = 2
```

```
c = 3
```

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

int main()
{
    // INT_MAX is the maximum
    // representable integer.
    int a = INT_MAX;

    printf("a = %i\n", a);
    a = a + 1;
    printf("a = %i\n", a);

    long int b = ((long int) INT_MAX)*2;
    printf("b = %li\n", b);
    return 0;
}
```

Be careful:

under/overflow

(e.g., **int** vs **long int**)

```
whovian@phy504:~/host/C$ nano overflow.c
whovian@phy504:~/host/C$ gcc overflow.c -o overflow
whovian@phy504:~/host/C$ ./overflow
```

a = 2147483647

a = -2147483648

b = 4294967294

Wrong result (overflow)

first cast to long then
multiply [Link on Under/Overflow](#)

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

```
int main()
{
    // INT_MAX is the maximum
    // representable integer.
    unsigned int a = INT_MAX;

    printf("a = %i\n", a);

    a = a + 1;

    printf("a = %u\n", a);

    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc overflow.c -o overflow
whovian@phy504:~/host/C$ ./overflow
a = 2147483647
a = 2147483648
whovian@phy504:~/host/C$
```

Signed (**int**) vs **unsigned int**

[Good discussion on StackOverflow](#)

Not having to carry a
bit for integer sign
double the range
without overflow

correct result

Be careful with **unsigned int**



Programmer Humor
@PROGRAMMERHUMOR

This for loop wreaked havoc in our code for some time [reddit.com/r/programmerhu...](https://www.reddit.com/r/programmerhumor/)

```
int main()
{
    unsigned int i = 0;

    for(i = 10; i >= 0; i--) {
        printf("%d\n", i);
    }

    return 0;
}
```

6:00 PM · 1/26/23 · 124K Views

This loop runs forever
(try it yourself)!

There is no negative
sign w/ **unsigned int**

When doing **i--** on **i=0**
iteration, the variable
underflows (assigned
garbage, but positive
garbage!)

Complex Numbers

```
GNU nano 4.8                                complex.c
#include <stdio.h>
#include <complex.h>

int main()
{
    double complex z1 = 1.0 + 3.0 * I;
    double complex z2 = 1.0 - 4.0 * I;

    double complex a = z1 + z2;
    printf("sum: Z1+Z2 = %f + %fi \n", creal(a), cimag(a));

    double complex b = z1 - z2;
    printf("diff: Z1-Z2 = %f + %fi \n", creal(b), cimag(b));

    return 0;
}
```



There is no %something for complex numbers in C

Assignment Operators

C allows joint arithmetic operators with assignment

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

```
int main (void)  
{
```

```
    int a = 10;
```

```
    printf ("a = %i\n", a);
```

```
    a += 10;
```

```
    printf ("a = %i\n", a);
```

```
    a *= 8;
```

```
    printf ("a = %i\n", a);
```

```
    a /= 4;
```

```
    printf ("a = %i\n", a);
```

```
    return 0;
```

```
}
```

a = a + 10

a = a * 8

a = a / 4

```
whovian@phy504:~/host/C$ nano aoper.c
```

```
whovian@phy504:~/host/C$ gcc aoper.c -o aoper
```

```
whovian@phy504:~/host/C$ ./aoper
```

```
a = 10
```

```
a = 20
```

```
a = 160
```

```
a = 40
```

Relational operators

Table 4.1 Relational Operators

Operator	Meaning	Example
<code>==</code>	Equal to	<code>count == 10</code>
<code>!=</code>	Not equal to	<code>flag != DONE</code>
<code><</code>	Less than	<code>a < b</code>
<code><=</code>	Less than or equal to	<code>low <= high</code>
<code>></code>	Greater than	<code>pointer > endOfList</code>
<code>>=</code>	Greater than or equal to	<code>j >= 0</code>

Important point: The relational operators have lower precedence than all arithmetic operators.

$a < b + c$

First, $b + c$ will be evaluated, then compared, i.e., the expression above is the equivalent to $a < (b + c)$

PS: $a \leq 200$ is equivalent to $a < 201$. I generally prefer *less than* or *greater than*, but with integers, they are all reasonable. Why?

Equality of float numbers is complicated. So, it is good practice to take $\leq$ and $\geq$ operators out of your mind when coding.

EXIT CODES

`stdlib.h` contains `EXIT_SUCCESS` and `EXIT_FAILURE` macros expand into integral expressions that can be used as arguments to the `exit` function

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

int main()
{
    printf("Hello World\n");
    return EXIT_SUCCESS;
}
```

```
whovian@phy504:~/host$ gcc exitsuccess.c -o exitsuccess
whovian@phy504:~/host$ ./exitsuccess
Hello World
whovian@phy504:~/host$ echo $?
0
```

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

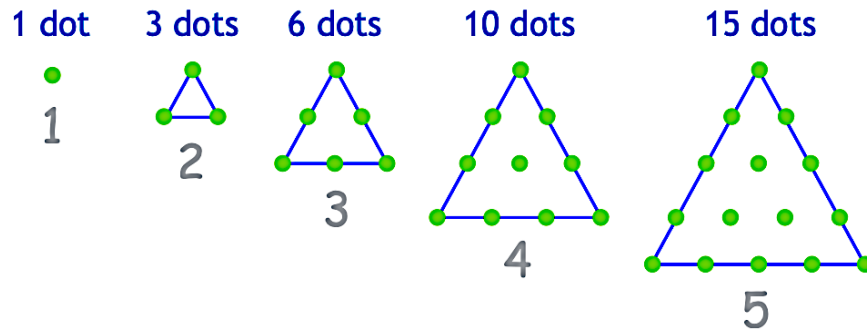
int main()
{
    printf("Ops, Failure\n");
    return EXIT_FAILURE;
}
```

```
whovian@phy504:~/host$ gcc exitfailure.c -o exitfailure
whovian@phy504:~/host$ ./exitfailure
Ops, Failure
whovian@phy504:~/host$ echo $?
1
```

good coding practice: use `EXIT_SUCCESS` and `EXIT_FAILURE`

Loops (**for**)

Triangular number sequence: 1, 3, 6, 10, 15, 21, 28, 36, 45, ...



This algorithm is simple to be implemented in a loop

By adding another row of dots and counting all the dots we can find the next number of the sequence.

```
whovian@phy504:~/host$ gcc trinumloop.c -o trinumloop
whovian@phy504:~/host$ ./trinumloop
The eighth triangular number is 36
```

GNU nano 4.8

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

```
int main ()
```

```
{
```

```
// compute the 8th triangular number
```

```
int x = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8;
```

```
printf ("The eighth triangular number is %i\n", x);
```

```
return 0;
```

```
}
```

C code without a loop is unfeasible for large values

PS: ok to declare and define in the same variable

```
// Program to calculate the 200th triangular number
```

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

```
int main (void)
{
```

```
    int x = 0; // SUPER IMPORTANT to initialize the variable
               // with zero EXPLICITLY
```

```
    for (int n = 1; n <= 200; n = n + 1)
    {
```

```
        // PS: you don't need {} for one-line for loops. However
        // I believe it is good practice to always include them
```

```
        x += n;
```

```
    }
```

```
    printf ("The 200th triangular number is %i\n", x);
```

```
// The issue of initialization
```

```
int y = 0;
```

```
int z; // This will have random garbage! Sometimes by chance z
       // will be zero (this is what we called undefined behavior
```

```
printf ("y and z = %i and %i\n", y, z);
```

```
return 0;
```

```
}
```

whovian@phy504:~/host\$ gcc trinumloop.c -o trinumloop

whovian@phy504:~/host\$./trinumloop

The 200th triangular number is 20100

y and z = 0 and 0

**don't let this
fool you**

**Good practice for multicore to
init variable **n** in the loop def**

Can this loop be threaded (multicore)? NO

Loops (**for**)

GNU nano 4.8

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

```
int main (void)
```

```
{ // Generate a table of triangular numbers
```

```
printf ("TABLE OF TRIANGULAR NUMBERS\n\n");
```

```
printf (" n      Sum from 1 to n\n");
```

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

```
int tn = 0;
```

I am using short name vars for display (but **NOT** ok in general!)

```
for (int n = 1; n < 11; ++n)
```

```
{
```

```
    tn += n;
```

```
    printf (" %2i      %i\n", n, tn);
```

```
}
```

```
return 0;
```

```
}
```

increment operator

This makes sure there is no shift between 9 and 10 (try without it)

Future: can't parallelize loops with **printf**

TABLE OF TRIANGULAR NUMBERS

n	Sum from 1 to n
---	----
1	1
2	3
3	6
4	10
5	15
6	21
7	28
8	36
9	45
10	55

Loops (**for**) with a touch with scanf scan for input

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

```
int main (void)
```

```
{
```

```
int m;
```

```
printf ("What triangular number do you want? ");
```

```
scanf ("%i", &m);
```

```
int tn = 0;
```

```
for (int n = 1; n < m+1; ++n)
```

```
{
```

```
tn += n;
```

```
}
```

```
printf ("Triangular number %i is %i\n", m, tn);
```

```
return 0;
```

```
}
```

```
whovian@phy504:~/host$ gcc printtrinumloop2.c -o printtrinumloop2
whovian@phy504:~/host$ ./printtrinumloop2
What triangular number do you want? 10
Triangular number 10 is 55
```

Reminder: don't use short cryptic names

(I am using here to fit code in a slide)

&x is the **reference** to variable x
(basically its memory position)
so scanf knows where to write it

Reminder: many books don't do that, but
you can (and should) define variables as
local as possible (here inside the loop)

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

Nested Loops (for)

```
int main (void)  
{
```

```
    for (int c=1; c<6; ++c)  
    {
```

```
        int m;
```



Be local: if you only need the variable inside the loop, declare it inside the loop!

```
        printf ("What triangular number do you want?");  
        scanf ("%i", &m);
```

```
        int tn = 0;
```



Be local: I only need **tn** below, I only declare **tn** here

```
        for (int n=1; n<m+1; ++n)
```

```
        {
```

```
            tn += n;
```

```
        }
```

```
        printf ("Triangular number %i is %i\n\n", m, tn);
```

```
    }
```

```
    return 0;
```

```
}
```

RUN THIS PROGRAM

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

```
int main (void)
```

```
{
```

```
    for (int n=0; n<5; ++n)
```

```
    {
```

```
        if (n==2)
```

```
        {
```

```
            break;
```

```
        }
```

```
        printf ("Testing Break  %i\n", n);
```

```
    }
```

```
    for (int n=0; n<5; ++n)
```

```
    {
```

```
        if (n==2)
```

```
        {
```

```
            continue;
```

```
        }
```

```
        printf ("Testing Continue  %i\n", n);
```

```
    }
```

```
    return 0;
```

```
}
```

break will exit the
loop completely

continue will skip
one iteration of the
loop

break / continue statements

```
whovian@phy504:~/hos
```

```
whovian@phy504:~/hos
```

```
Testing Break  0
```

```
Testing Break  1
```

```
Testing Continue  0
```

```
Testing Continue  1
```

```
Testing Continue  3
```

```
Testing Continue  4
```

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

```
int main (void)
```

```
{
```

```
    for (int n=0; n<3; ++n)
```

```
    {
```

```
        for (int m=0; m<3; ++m)
```

```
        {
```

```
            if (m==1)
```

```
            {
```

```
                break;
```

```
            }
```

```
            printf ("Testing Break n=%i m=%i\n", n, m);
```

```
        }
```

```
    }
```

```
    for (int n=0; n<3; ++n)
```

```
    {
```

```
        for (int m=0; m<3; ++m)
```

```
        {
```

```
            if (m==1)
```

```
            {
```

```
                continue;
```

```
            }
```

```
            printf ("Testing Continue n=%i m=%i\n", n, m);
```

```
        }
```

```
    }
```

```
    return 0;
```

```
}
```

break will exit only
the inner loop

continue will only
skip one iteration of
the inner loop

break / continue statements

```
whovian@phy504:~/host$ ./
Testing Break n=0 m=0
Testing Break n=1 m=0
Testing Break n=2 m=0
Testing Continue n=0 m=0
Testing Continue n=0 m=2
Testing Continue n=1 m=0
Testing Continue n=1 m=2
Testing Continue n=2 m=0
Testing Continue n=2 m=2
```

Compiler is your friend

You can ask the compiler to
be extra careful

Compiler warnings can be life
saving

You should aim to
resolve/understand all
warnings from your code

My compiler watching me continue to
code, ignoring its warnings:



The things my compiler sees..

- **-Wall** — There is high confidence in both the value and low false-positive rate of these warnings
- **-Wextra** — These warnings are believed to be valuable and robust (not buggy), but they may have high false-positive rates or common philosophical objections
- **-Wpedantic** — Enforces strict ISO C/C++.

Homework I

- Run the “Hello World” program on your system. Experiment with omitting parts of the program to see what error messages you get.
- Write a program that prints the following text
 - In **C**, lowercase letters are significant;
 - **main()** is where program execution begins;
 - Opening and closing braces enclose program statements in a routine;
 - All program statements must be terminated by a semicolon.
- Write a program that uses a **printf** statement to print out a float variable using two decimal places.
- What is the output of the following C code?
If there is a problem, fix this code.

```
#include <stdio.h>

int main()
{
    int i = 500, j = 700;
    printf("%d %d\n");
    return 0;
}
```

Homework II

- Identify the syntactic errors in the following program. Then type in and run the corrected program to ensure you have correctly identified all the mistakes

```
#include <stdio.h>

int main (Void)
(
    INT    sum;
    /* COMPUTE RESULT
    sum = 25 + 37 - 19
    /* DISPLAY RESULTS //
    printf ("The answer is %i\n" sum);
    return 0;
}
```

Practice doing that without the compiler (be your own compiler)!

Homework III

- Write a program that evaluates the following expression and displays the results (remember to use exponential format to display the result)

$$(3.31 \times 10^{-8} \times 2.01 \times 10^{-7}) / (7.16 \times 10^{-6} + 2.01 \times 10^{-8})$$

- Write a program that subtracts the value 15 from 87 and displays the result, together with an appropriate message, at the terminal
- Write a program that converts 27 degrees Fahrenheit (F) to degrees Celsius (C) the following formula $C = 5(F - 32) / 9$
- Write a program to evaluate the polynomial shown here $3x^3 - 5x^2 + 6$ for $x = 2.55$ (and display the results in scientific notation)
- Explain the difference between (test your theory!)

```
printf ("Result 1 = %i \n", (int) 29.55 + (int) 21.99);  
printf ("Result 2 = %i \n", (int) round(29.55) + (int) round(21.99));
```

Homework IV

- c) Suppose f is a floating-point variable whose value is 5.5 Print the value of `((int)f)%2` as an integer.
- d) Suppose an integer variable i has a value 1. What is printed below:

```
printf("i = %d \n", i);  
printf("i = %d \n", ++i);  
printf("i = %d \n", i);
```

If the print commands had been

```
printf("i = %d \n", i);  
printf("i = %d \n", i++);  
printf("i = %d \n", i);
```

Homework V

5. A C program contains the following declarations and initial assignments:

```
int i = 8, j = 5;  
double x = 0.005, y = -0.01;  
char c = 'c', d = 'd';
```

Determine the value of each of the following expressions and explain the meaning of the library function:

- a) `abs(i - 2 * j)`
- b) `fabs(x+y)`
- c) `isprint(c)`
- d) `isdigit(c)`
- e) `toupper(d)`
- f) `ceil(x)`
- g) `ceil(x+y)`
- h) `floor(x)`
- i) `floor(x+y)`
- j) `islower(c)`

Homework VI

- C has a nice extension named VLA (Variable Length Arrays). This is an array whose size is determined at runtime. Using VLA, write **twoSum** function searches for two indices such that their corresponding elements sum to the target. This function should have the following profile (you are going to need for loops)

void twoSum(const int nums[], int n, int target, int result[2]);

The main should look like this

```
int main() {
    int nums[] = {2, 7, 11, 15};
    int n = sizeof(nums) / sizeof(nums[0]);
    int target = 9;

    // Using a VLA (in this case, fixed-size) for the result.
    int result[2];

    twoSum(nums, n, target, result);

    if (result[0] != -1) {
        printf("Indices: [%d, %d]\n", result[0], result[1]);
    } else {
        printf("No two-sum solution found.\n");
    }

    return 0;
}
```

Homework VII

- Using VLA, write a function that returns the Median of two Sorted Arrays. The function should look like

double MSA(const int nums1[], int m, const int nums2[], int n);

One approach would look like

```
#include <stdio.h>

// nums1 and nums2 are input arrays of sizes m and n respectively.
double MSA(const int nums1[], int m, const int nums2[], int n)
{
    // Create a VLA to hold the merged elements.
    (...)

    // Merge the two sorted arrays.
    (...)

    // Copy any remaining elements from nums1.
    (...)

    // Copy any remaining elements from nums2.
    (...)

    int total = m + n;
    // If the total number of elements is odd, return the middle element.
    (...)
}

int main(void)
{
    int nums1[] = {1, 3, 5};
    int nums2[] = {2, 4, 6, 8};

    // Compute sizes of the arrays.
    int m = sizeof(nums1) / sizeof(nums1[0]);
    int n = sizeof(nums2) / sizeof(nums2[0]);

    printf("Median is: %f\n", MSA(nums1, m, nums2, n));
    return 0;
}
```