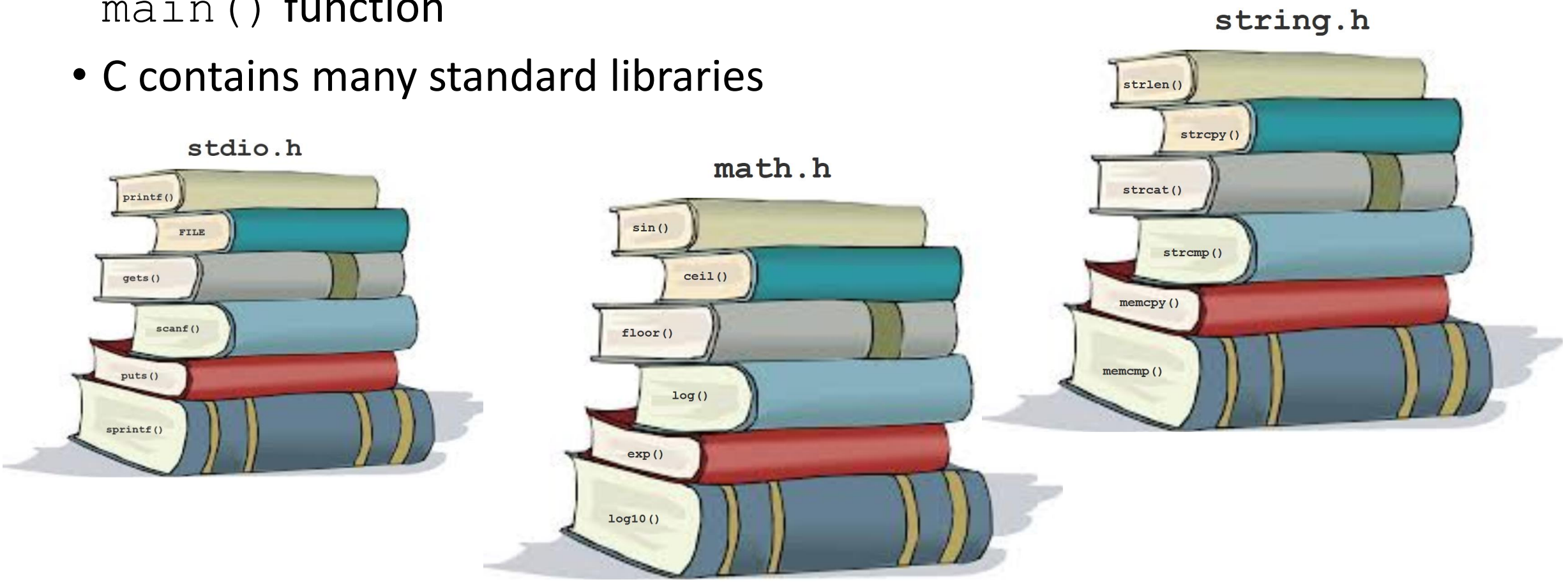
CSE 1320

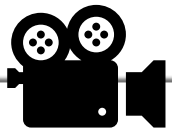
Week of 01/30/2023

Instructor : Donna French

Libraries in C

- A library in C is a collection of functions and definitions without a `main()` function
- C contains many standard libraries





```c
/* Donna French 1000074079 */

/* This is my first C program for CSE 1320 */

#include <stdio.h>

void GoodbyeWorld(void)
{
    printf("Goodbye World");
}

int main(void)
{
    printf("Hello World");

    GoodbyeWorld();

    return 0;
}

```

EscapeDemo.c
BackspaceDemo.c

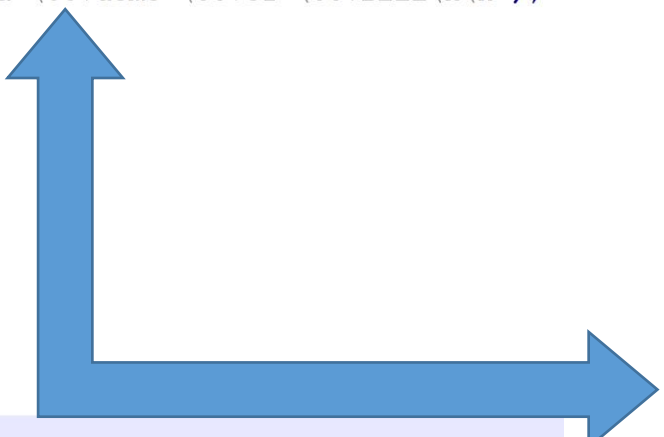
An Introduction to Output in C

Other **Escape** Characters like \n

\t	the tab character
\b	the backspace character
\"	the double quote character in a string
\'	the single quote character
\\	the backslash character
\0	the null character

```c
/* Backspace Demo */

#include <stdio.h>

void BackspaceDemo(void)
{
    printf("My name is Ariel");
    fflush(NULL);
    sleep(5);
    printf("\b\b\b\b\b");
    printf("Erik");
    printf("\n");

    /* Pause execution until a key is pressed */
    puts(" ");
    getchar();

    printf("\n\nThe password is \"ASECRET\"");
    fflush(NULL);
    sleep(5);
    printf("\b\b\b\b\b\b\b\b       \n\n\n\n");
}

int main (void)
{
    BackspaceDemo();

    return 0;
}
```

```c
/* Demonstration of escape characters */

#include <stdio.h>

void DemoTab(void)
{
    printf("This\tis\ta\tdemo\tof\tTAB\n\n\n");
}

void DemoBackspace(void)
{
    printf("This\bis\ba\bdemo\bof\bBACKSPACE\n\n\n");
}

void DemoDoubleQuote(void)
{
    printf("This\"is\"a\"demo\"of\"DOUBLEQUOTE\n\n\n");
}

void DemoSingleQuote(void)
{
    printf("This\'is\'a\'demo\'of\'SINGLEQUOTE\n\n\n");
}

void DemoBackSlash(void)
{
    printf("This\\is\\a\\demo\\of\\BACKSLASH\n\n\n");
}

void DemoBELL(void)
{
    printf("This \007is \007a \007demo \007of \007BELL\n\n");
}

int main (void)
{
    DemoTab();
    DemoBackspace();
    DemoDoubleQuote();
    DemoSingleQuote();
    DemoBackSlash();
    DemoBELL();

    return 0;
}
```



frenchdm@omega:~

```
[frenchdm@omega ~]$
```

Ascii	Char
0	Null
1	Start of heading
2	Start of text
3	End of text
4	End of transmit
5	Enquiry
6	Acknowledge
7	Audible bell

\110\145\154\154\157\041\012

Using ASCII to print symbols

Hello!

Terminal - student@maverick: /media/sf_VM/CSE1320

File Edit View Terminal Tabs Help

```
student@maverick:/media/sf_VM/CSE1320$
```

ASCII octal values

Used !. to repeat the last command that started with .

Variables

Variables are used to hold data while a program is executing.

- Variable must be declared before it can be used
- Declaration establishes the variable's name and type
- Compiler reserves space in memory for each variable

Input and Output with Variables

Function `scanf()` scans for formatted input.

```
scanf( control_string, args, … )
```

scans input for characters requested by conversion specifications in the `control_string`. For each conversion specification in the `control_string`, attempts to convert the input value and store in the corresponding argument.

```
printf("Enter your favorite number\n");
scanf("%d", &FavNum);
printf("Your favorite number is %d\n", FavNum);
```

& means the address of the variable `FavNum`

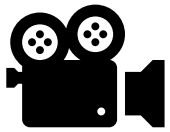
location in pantry

Input and Output with Variables

scanfDemo.c

```
#include <stdio.h>

int main (void)
{
     int FavNum;

     printf("Enter your favorite number\n");
     scanf("%d", &FavNum);
     printf("Your favorite number is %d\n", FavNum);

     return 0;
}
```

& means the address of `FavNum`



What happens when you use

```
scanf("%d\n", &x);
```

RemainderDemo.c
UnaryDemo.c

Arithmetic Operators

addition and unary plus	+ and ++
unary minus and subtraction	-- and -
multiplication	*
division	/
remainder	%

- Integer division
  - Truncating division
    - Fractional part (remainder) is truncated
- Remainder (MOD)
  - Returns the remainder from division
- Unary ++ and --

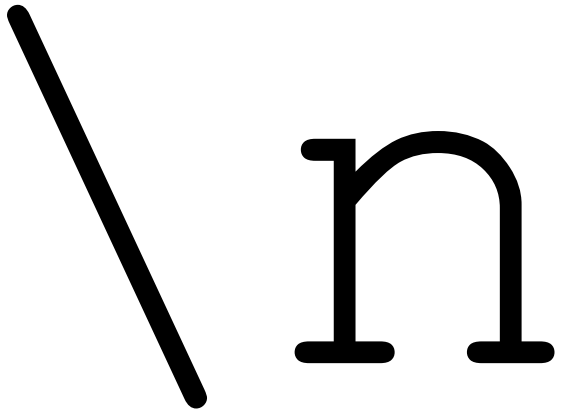
```c
/* Unary Demo */

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

int main (void)
{
    int counter;

    printf("Enter a positive value for counter ");
    scanf("%d", &counter);

    printf("%d is the negative value of %d\n",
           -counter, counter);

    counter = -counter;
    printf("\n\nThe negative value of counter is %d\n", counter);

    counter = +counter;
    printf("\n\nThe positive value of counter is %d\n", counter);

    counter = abs(counter);
    printf("\n\nThe positive value of counter is %d\n", counter);

    return 0;
}
```

```
Enter a positive value for counter 12
-12 is the negative value of 12


The negative value of counter is -12


The positive value of counter is -12


The positive value of counter is 12
```

```c
/* Remainder Demo */

#include <stdio.h>

int main (void)
{
    int divisor;
    int dividend;
    int quotient;
    int remainder;

    printf("Enter the divisor ");
    scanf("%d", &divisor);

    printf("Enter the dividend ");
    scanf("%d", &dividend);

    quotient = dividend / divisor;
    printf("\nThe quotient is %d\n", quotient);

    remainder = dividend % divisor;
    printf("\nThe remainder is %d\n", remainder);

    return 0;
}
```

```
Enter the divisor 5
Enter the dividend 100

The quotient is 20

The remainder is 0
```

```
Enter the divisor 5
Enter the dividend 101

The quotient is 20

The remainder is 1
```



vs

scanf() and \n

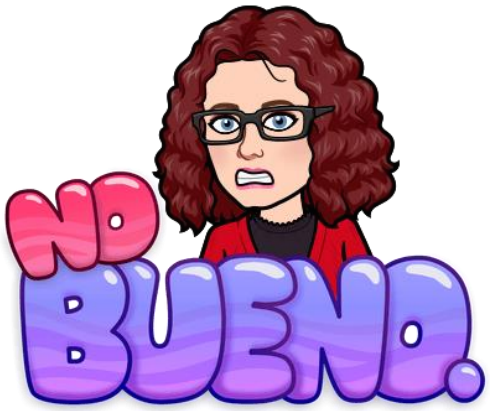
```
#include <stdio.h>

int main(void)
{
  int MyNum = 0;
  printf("Enter a number: ");
  scanf("%d \n", MyNum);
  printf("The entered number is %d, MyNum");
  return 0;
}
```

Programmer Joke

What is the best prefix for a global variable?

//

Global variables are not allowed in Coding Assignments or Quizzes unless specifically included as part of an assignment.



Increment/Decrement Unary Operators

++

Increment Operator

Add 1 to a variable

Two forms

```
i++
++i
```

--

Decrement Operator

Subtract 1 from a variable

Two forms

```
i--
--i
```

Arithmetic Operators

Precedence of Arithmetic Operations

High precedence

- Unary operators ++,--
- Multiplicative Operators *, /, %
- Additive Operators +, -
- Assignment Operators =

Low precedence

Within each group, the operations associate from left to right.

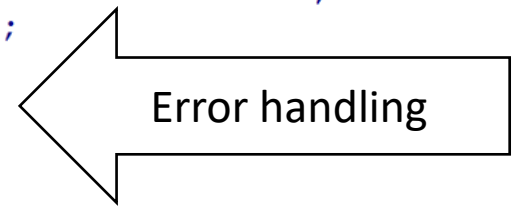
```
a*b/c = (a * b) /c
```

```
2*a+4/b = (2*a) + (4/b)
```

Structured Programming in C

- Write source code that is
  - modular
  - easily modifiable
  - robust (handles errors gracefully)
  - readable
- Write functions that can be used with little or no modification in many programs
- Write functions to do one task that is not too long and can be understood easily

```
189         opn_files ();
190         printf ("\nProcess invoices for %-4.4s\n",whse);
191
192         /* Find orders and write to gszXMLbuff */
193         get_ords ();
194
195         SORTMERGEFINISH ((short *)scb,1);    /*    Finish the sort process  */
196
197         /* Write gszXMLbuff to transmit file */
198         if (cnt && glTotalFileBytes)  /*  Were there any invoices?  */
199         {
200            /* Create the file and open it */
201            create_xmit_file (szInvoiceFile, dataset,gszWhse , "X");
202            nError = FILE_OPEN_(szInvoiceFile,(short)strlen(szInvoiceFile), &fd, ,,);
203            if(nError)
204            {
205               sprintf(gszMsg,"Error %d trying to open Invoice file %s",
206                                 nError, szInvoiceFile);
207               fnProcessError();
208               SENDEMAIL((short *)&gstErrorEmail);
209               msgabend (gszMsg, (short)nError, 0);
210            }
211
212            /* Write gszXMLbuff to the file */
213            if( glTotalFileBytes <= BYTES_TO_WRITE)
214            {
215               if ( nError = DISCWRITE(fd, (short *)&gszXMLbuff, (short)glTotalFileBytes))
216               {
217                  sprintf(gszMsg,"Error %d trying to write to %s file",
218                                 nError, szInvoiceFile);
219                  fnProcessError();
220                  SENDEMAIL((short *)&gstErrorEmail);
221                  FILE_CLOSE_(fd);
222                  msgabend (gszMsg, (short)nError, 0);
223               }
224            }
225            else
226            {
```



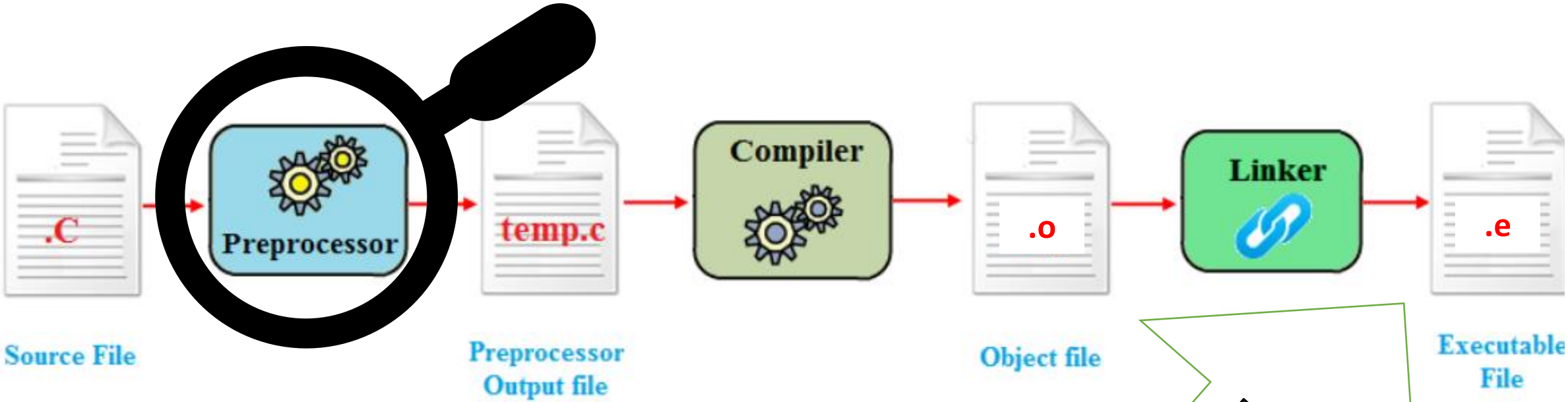
Required Formatting of Code

The opening brace for a function should be given its own line and the closing brace should line up with the opening brace.

Any code lines within the braces should be indented the same amount which must be between 3 and 5 spaces.

```
int main(void)
{
    my first line
    my second line
    my third line
    return 0;
}
```

From .c source file to executable



We will talk about the compiler and linker when we get to makefiles.

For now, you only see `a.out` as the executable

```
/* Donna French 1000074079 */

/* This is my first C program for CSE 1320 */

#include <stdio.h>

int main(void)
{
    printf("Hello World\n");

    return 0;
}
```

Preprocessor

The C preprocessor executes before a program is compiled.

Some actions it performs are the **inclusion of other files in the file being compiled**, definition of symbolic constants and macros, conditional compilation of program code and conditional execution of preprocessor directives.

Preprocessor directives begin with # and only whitespace characters and comments may appear before a preprocessor directive on a line.

Preprocessor directive `#include`

The `#include` directive causes a copy of a specified file to be included in place of the directive.

Preprocessor directive `#include`

The two forms of the `#include` directive are

```
#include <filename>
#include "filename"
```

The difference between these is the location the preprocessor begins searches for the file to be included.

Preprocessor directive `#include`

```
#include <filename>
```

If the file name is enclosed in angle brackets (< and >)—used for standard library headers—the search is performed in an implementation-dependent manner, normally through predesignated compiler and system directories.

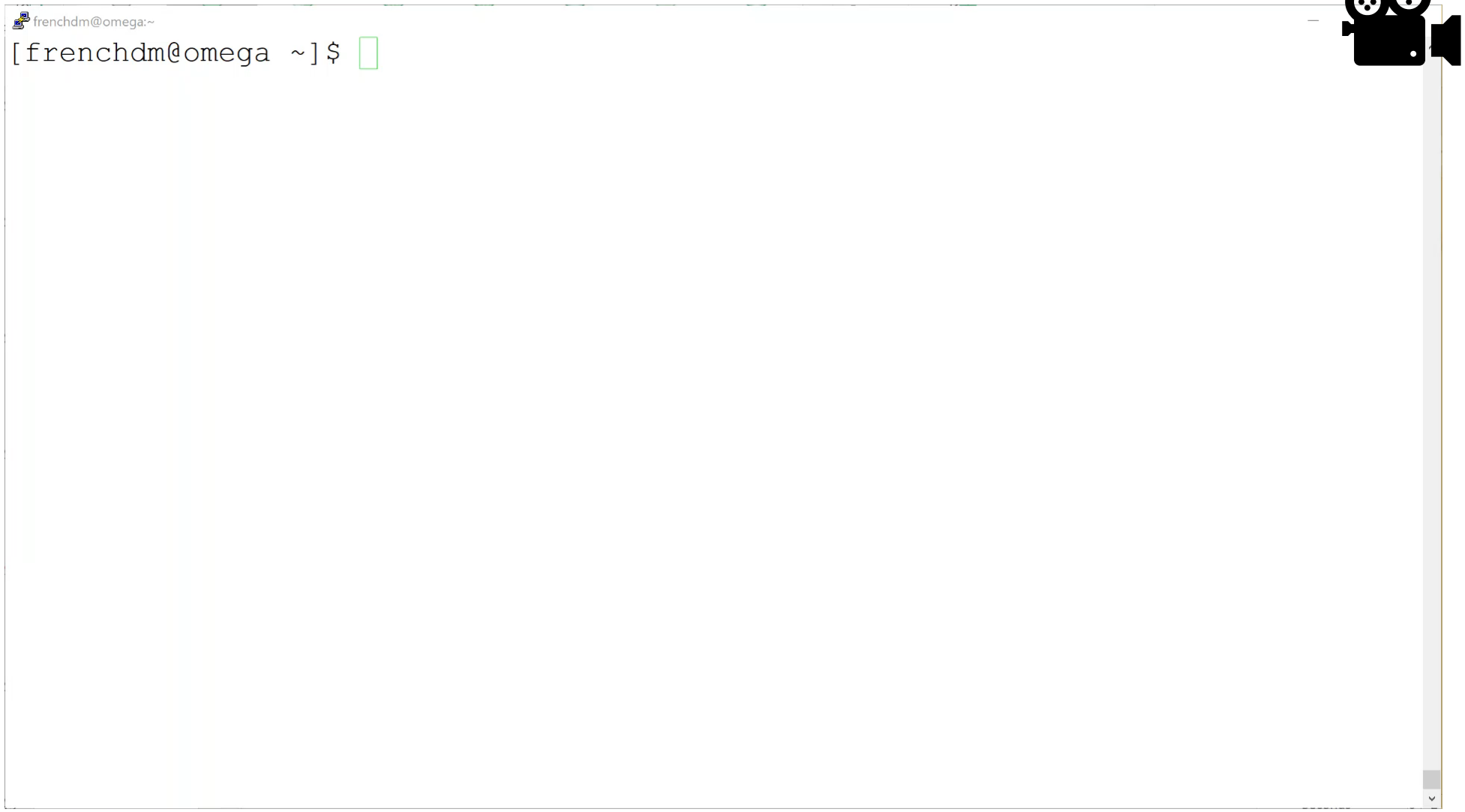
```
#include "filename"
```

If the file name is enclosed in quotes, the preprocessor starts searches in the same directory as the file being compiled for the file to be included (and may search other locations, too).

Using your own header file for an `#include`

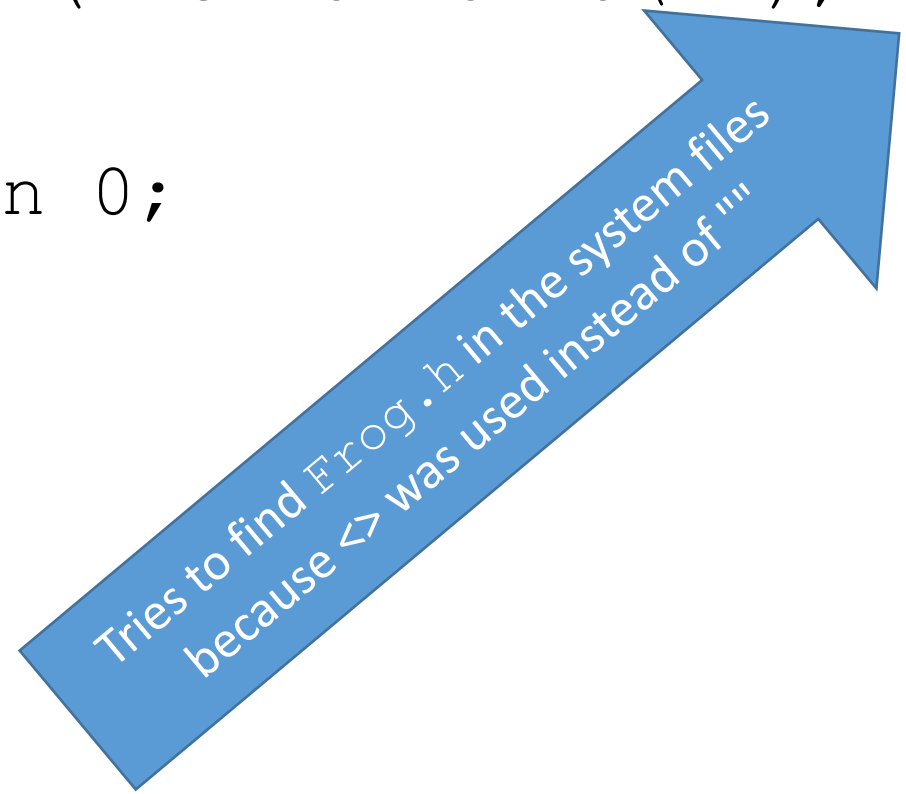
Use
" "

Not
<>



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


int main(void)
{
    printf("Hello World\n");

    return 0;
}
```

```
[frenchdm@omega ~]$ gcc
HelloWorld.c -E
# 1 "HelloWorld.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "HelloWorld.c"
HelloWorld.c:6:18: error:
Frog.h: No such file or
directory


int main(void)
{
    printf("Hello World\n");

    return 0;
}
```



`#define` Preprocessor Directive Symbolic Constants

The `#define` directive creates symbolic constants—constants represented as symbols—and macros—operations defined as symbols.

Same concept as `final` constants in Java

The `#define` directive format is

```
#define  identifier  replacement-text
```

When this line appears in a file, all subsequent occurrences of identifier that do not appear in string literals will be replaced by replacement-text automatically before the program is compiled.

`#define` Preprocessor Directive Symbolic Constants

```
#define PI 3.14159
```

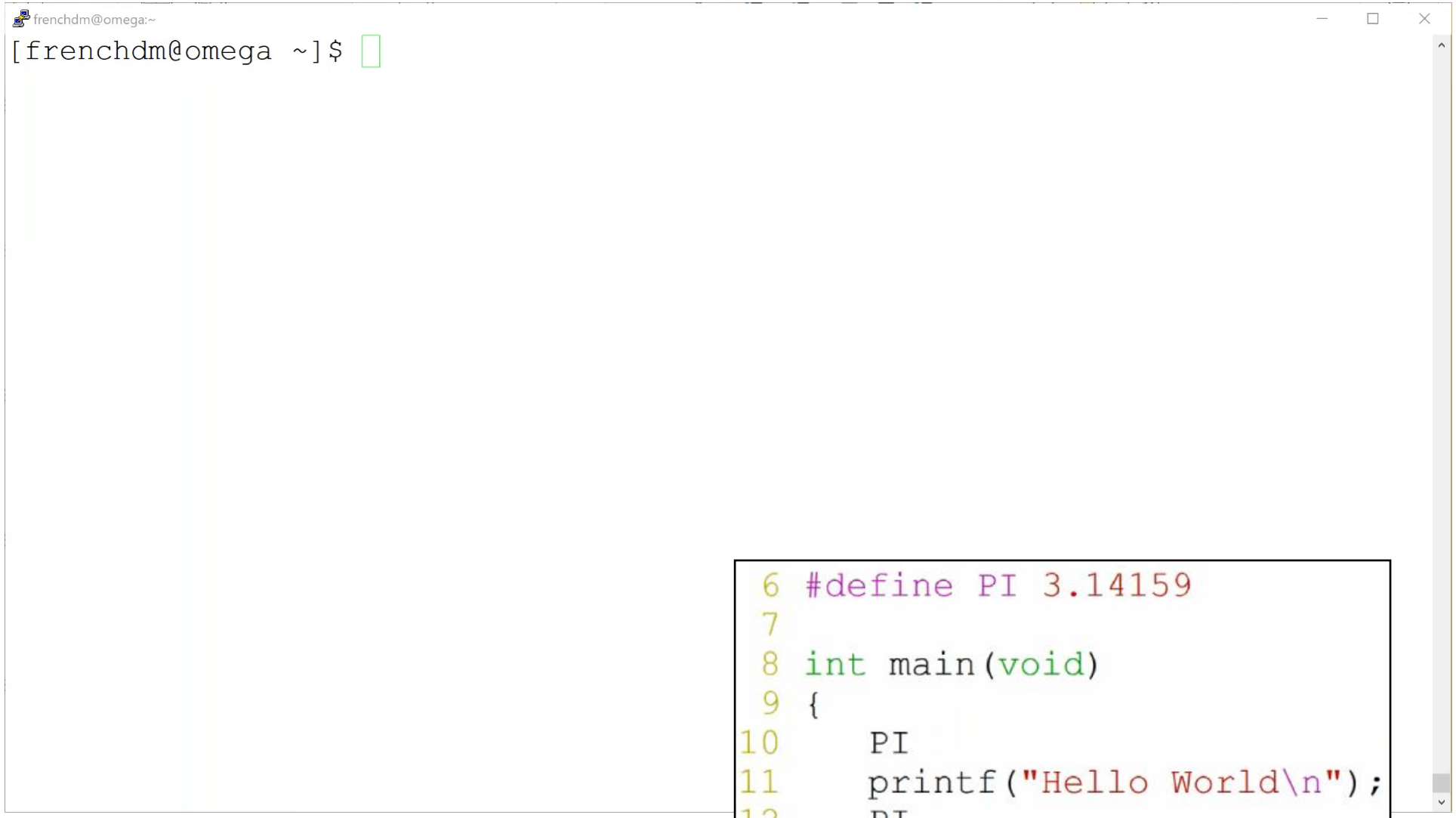
Anywhere PI is used in the program, the preprocessor will replace all subsequent occurrences of the symbolic constant PI with the numeric constant 3.14159.

After setting the `define`, everywhere that references

`PI`

is changed to

`3.14159`




```
 6 #define PI 3.14159
 7
 8 int main(void)
 9 {
10     PI
11     printf("Hello World\n");
12     PI
13     return 0;
14     PI
15 }
```

`#define` Preprocessor Directive
Symbolic Constants

Symbolic constants enable you to create a name for a constant and use the name throughout the program.

If the constant needs to be modified throughout the program, it can be modified *once* in the #define directive.

When the program is recompiled, all occurrences of the constant in the program will be modified accordingly.

`#define` Preprocessor Directive Symbolic Constants

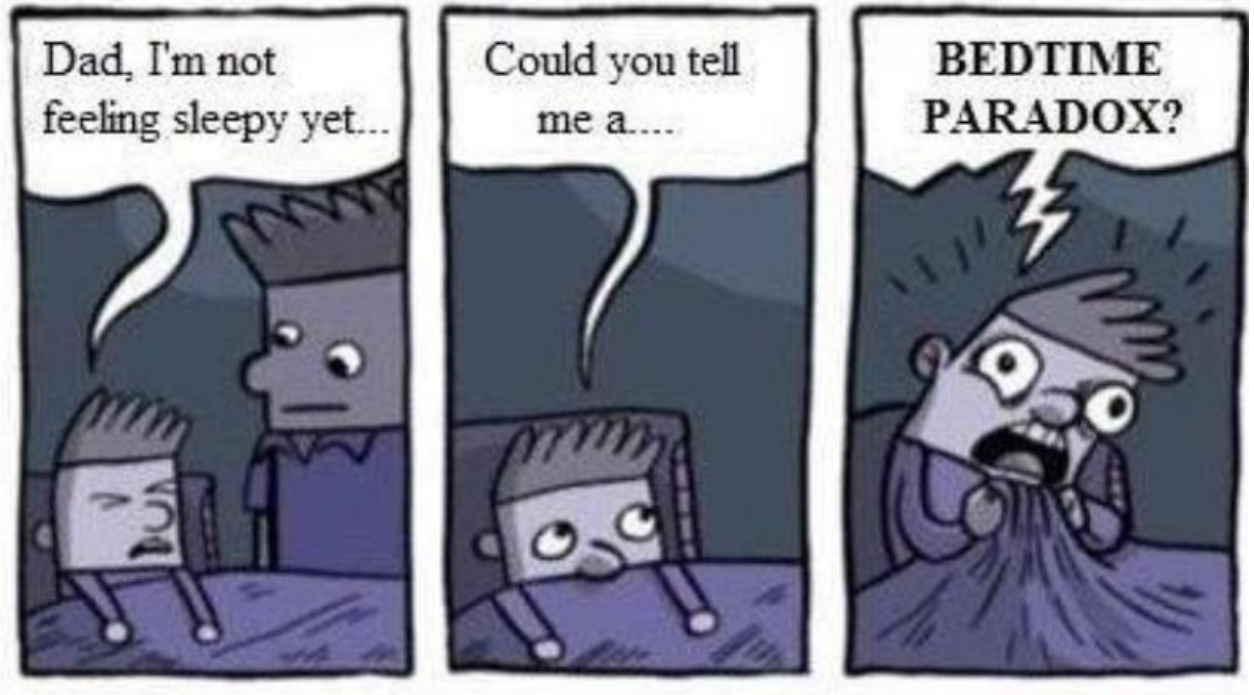
- adds to a program's readability
- allows a program to be more easily modified

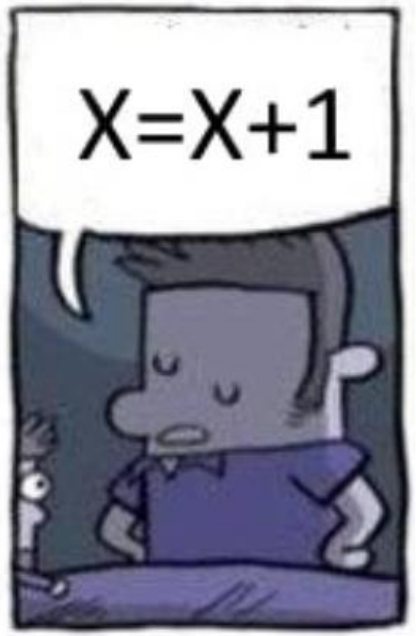
Note - `#define` in C does not use an = sign

```
#define SUNDAY    0
#define MONDAY    1
#define TUESDAY   2
#define WEDNESDAY 3
#define THURSDAY  4
#define FRIDAY    5
#define SATURDAY  6
```

```
#define BYTES_TO_WRITE 1096
#define NBR_STATS 3
#define ORDHDR_TEMPONBR_KEY 'TN'
#define FALSE 0
#define TRUE 1
```

```
225             else
226             {
227                 cPtr = gszXMLbuff;
228                 nBytesTowrite = BYTES_TO_WRITE;
229                 while (*cPtr)
230                 {
231                     if( (long)strlen(cPtr) > BYTES_TO_WRITE)
232                     {
233                         nBytesTowrite = BYTES_TO_WRITE;
234                     }
235                     else
236                     {
237                         nBytesTowrite = (short)strlen(cPtr);
238                     }
```


Dad, I'm not feeling sleepy yet...
Could you tell me a....
BEDTIME PARADOX?


X=X+1

Expressions vs Statements

Expressions

sequences of tokens that can be evaluated to a numerical quantity

- can be a single number
- can be an identifier
- can be more complicated sequence of tokens
- can contain any of the operators in C
- arguments to functions

Statement

sequence of tokens terminated with a semicolon that can be recognized by the compiler

- may not have values
- purpose might be to select which set of statements to execute in a given circumstance
- purpose might be to cause a sequence of statements to be executed more than once (control statement)
- cannot be an argument to a function

Expressions vs Statements

Expressions

something you can print or assign to a variable

Statement

everything else that is not an expression.

lvalue vs rvalue Expressions

lvalue

- an expression that has a location in memory such as a name of a variable
- expressions whose values can be either changed or evaluated
- **used on the left side of an assignment statement**

rvalue

- can be evaluated but cannot be changed
  - single character token ‘5’
- **used on the right side of an assignment statement**

lvalue vs rvalue Expressions

```
int x;
int y;
```

Expression	lvalue	rvalue
`x`	yes	yes
`x + 3`	no	yes
`y`	yes	yes
`2*y - 7`	no	yes
`(-2/y + 7 % x)`	no	yes

Assignment Expression

```
expr1 = expr2
```

`expr1` is an lvalue `expr2` is an rvalue

When this assignment expression is evaluated, `expr2` is fully evaluated before the assignment expression itself takes on that value

```
x = 5
```

lvalue on left side indicates where to store the value obtained from the evaluation of the expression on the right side.

Assignment Expression

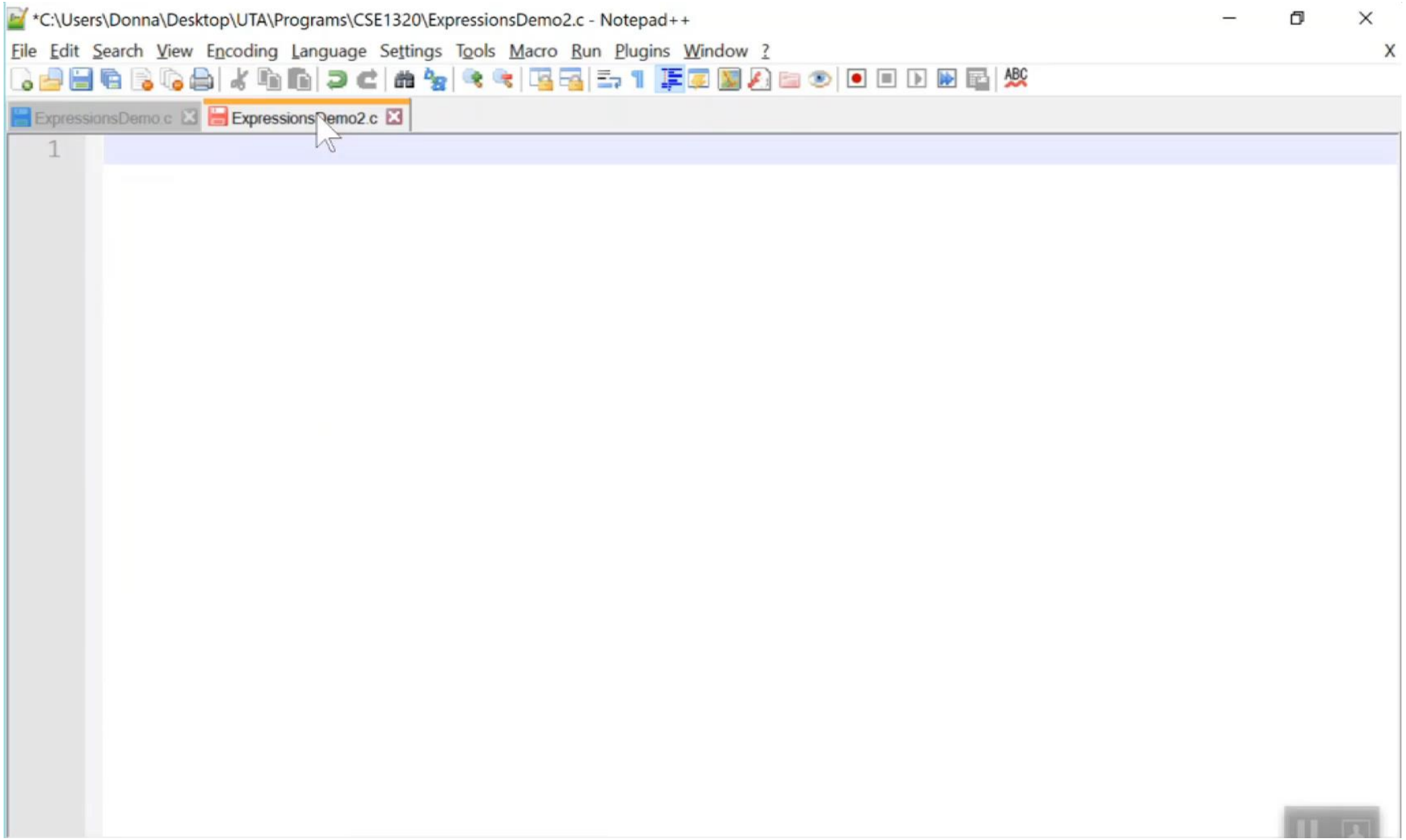
```
int x = 1;
int y = 1;
int z = 1;
```

Given these variables, which statements are valid?

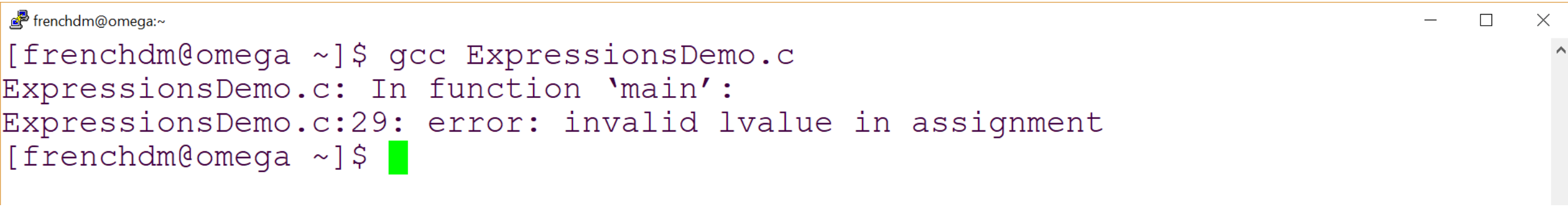
```
z = y = 4*x + 5;
```

```
z + 1 = y = 4*x + 5;
```

```
z = y + 1 = 4*x + 5;
```


*C:\Users\Donna\Desktop\UTA\Programs\CSE1320\ExpressionsDemo2.c - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
ExpressionsDemo.c
ExpressionsDemo2.c
1

$$y = x + 2 - 3 = z$$



ExpressionsDemo.c

```
frenchdm@omega:~
[frenchdm@omega ~]$ gcc ExpressionsDemo2.c
ExpressionsDemo2.c: In function ‘main’:
ExpressionsDemo2.c:15: error: invalid lvalue in assig
nment
[frenchdm@omega ~]$
```

*C:\Users\Donna\Desktop\UTA\Programs\CSE1320\ExpressionsDemo2.c - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

ExpressionsDemo.c | ExpressionsDemo2.c

```c
// Expression Demo 2

#include <stdio.h>

int main(void)
{
    int x = 1;
    int y;
    int z = 10;

    printf("The value is %d\n", x);
    printf("The value is %d\n", x+2);
    printf("The value is %d\n", x+2-3);
    printf("The value is %d\n", y = x+2-3);
    printf("The value is %d\n", y = x+2-3 = z);

    return 0;
}
```

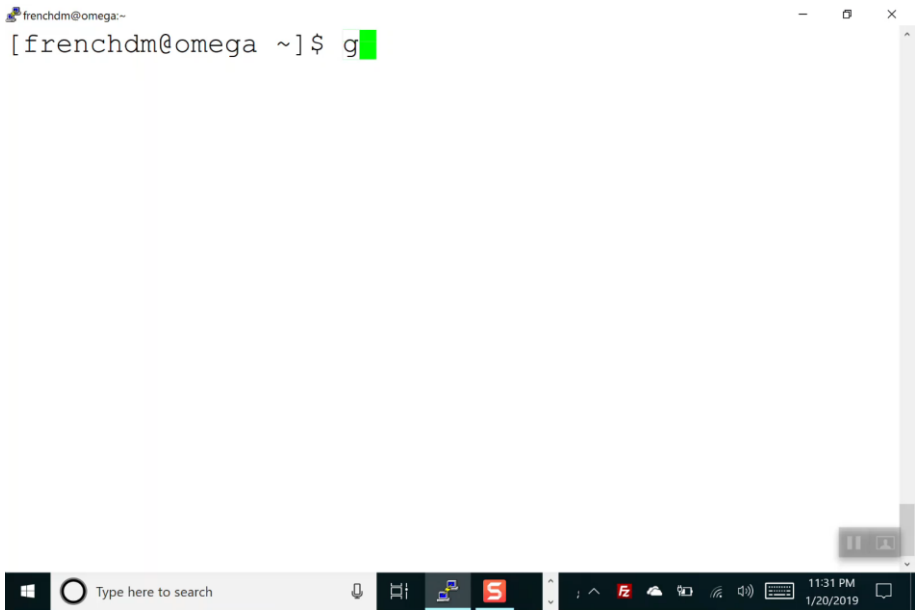
y = x+2-3 = z

Blocks/Compound Statements

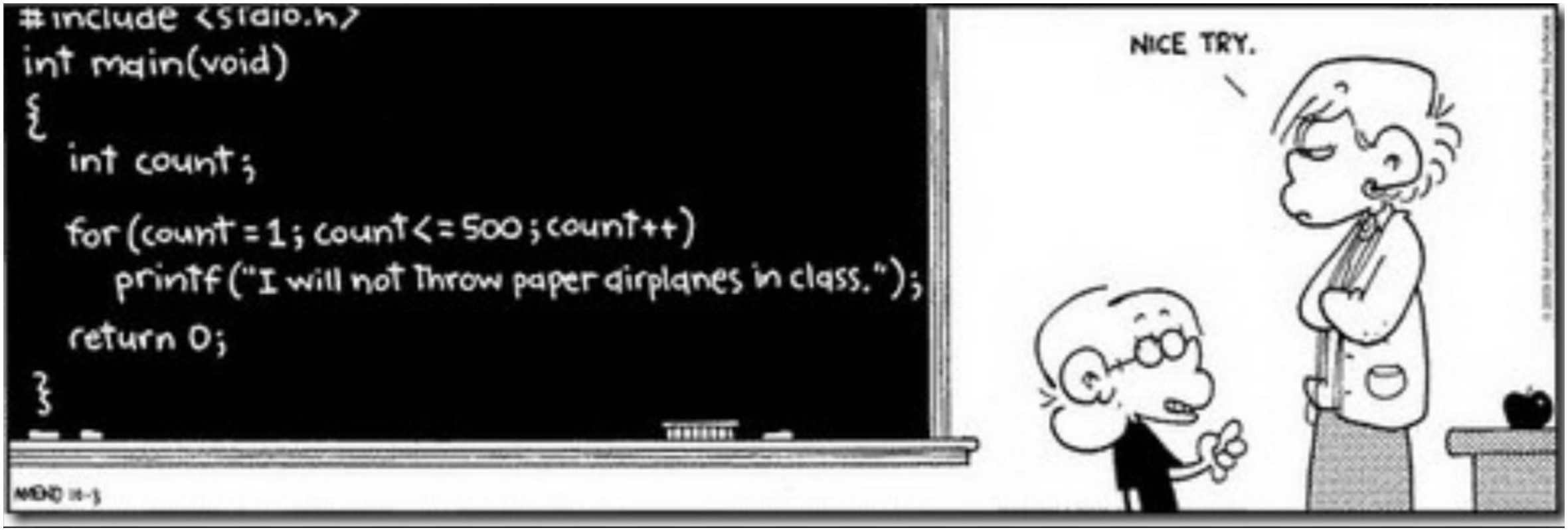
- a set of statements contained within a pair of braces {}
- sequence of statements that can be used anyplace in the syntax that a simple statement can be used
- Conditional and loop structures use blocks to force multiple statements to be executed
- Use caution declaring variables within a block – scope rules apply

BlockDemo.c

```
int main (void)
{
   int VarLocalToMain = 2;

   {
      int VarLocalToBlock = 3;
      printf("Value of VarLocalToMain is %d\n",
             VarLocalToMain);
      printf("Value of VarLocalToBlock is %d\n",
             VarLocalToBlock);
   }

   printf("Value of VarLocalToMain is %d\n", VarLocalToMain);
   printf("Value of VarLocalToBlock is %d\n", VarLocalToBlock);

   return 0;
}
```



The for Loop


#include <stdio.h>
int main(void)
{
int count;
for (count = 1; count <= 500; count++)
printf ("I will not throw paper airplanes in class.");
return 0;
}
NICE TRY.

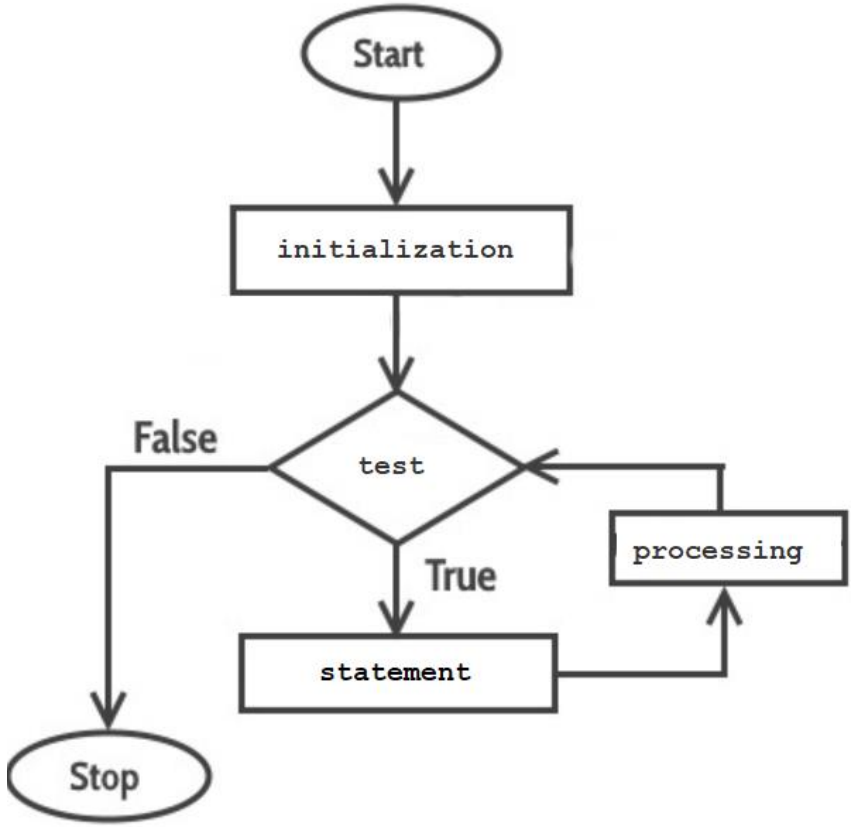
The for Loop

```
for (initialization; test; processing)
    statement
```

`initialization`
- expression that is evaluated once as the loop is entered

`test`
- expression that is evaluated as a condition for continuing the loop
  - if the value of `test` is nonzero, statement is executed
  - if the value of `test` is zero, the execution of the for loop is terminated

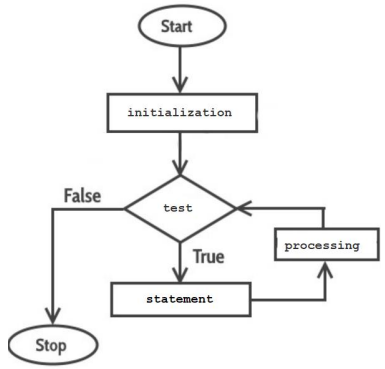
`processing`
- expression that is evaluated after statement is executed each time through the loop
- does bottom-of-the-loop processing



any or all of the three expressions may be omitted

forloopDemo.c

The `for` Loop




```
int i;

for (i = 0; i <= 3; i++)
   printf("i = %d\n", i);
```

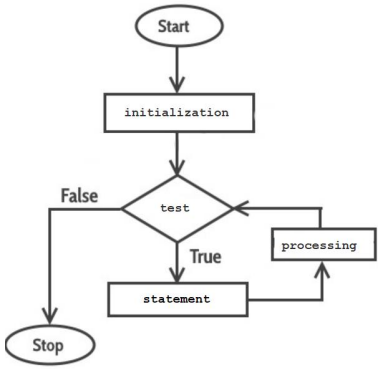
```
i = 0
i = 1
i = 2
i = 3
```

```
int i;

for (i = -3; i <= 3; i++)
   printf("i = %d\n", i);
```

```
i = -3
i = -2
i = -1
i = 0
i = 1
i = 2
i = 3
```

forloopDemo.c

The `for` Loop

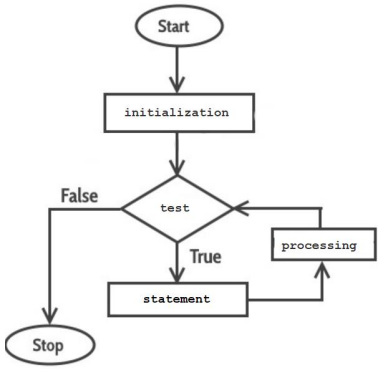



```
int i;

for (i = 4; i > 0; i--)
   printf("i = %d\n", i);

i = 4
i = 3
i = 2
i = 1
```

```
int i;

for (i = 2; i > -2; i--)
   printf("i = %d\n", i);

i = 2
i = 1
i = 0
i = -1
```

forloopDemo.c

The `for` Loop




```
int i;

for (i = 1; i < 16; i+=4)
   printf("i = %d\n", i);

i = 1
i = 5
i = 9
i = 13
```

```
int i;

for (i = 16; i > 0; i/=3)
   printf("i = %d\n", i);

i = 16
i = 5
i = 1
```

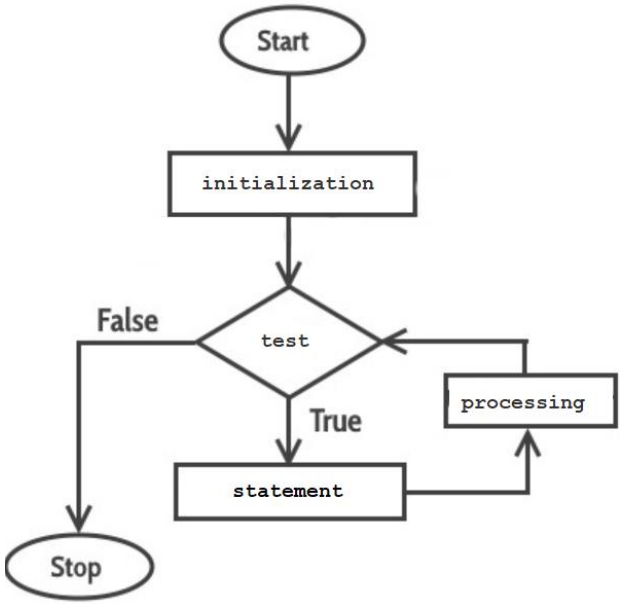
The `for` Loop

```
for (i = 1; i < 10; i+=3)
printf("i = %d\n", i);
printf("\n\ni = %d\n", i);
```

```
i = 1
i = 4
i = 7

i = 10
```

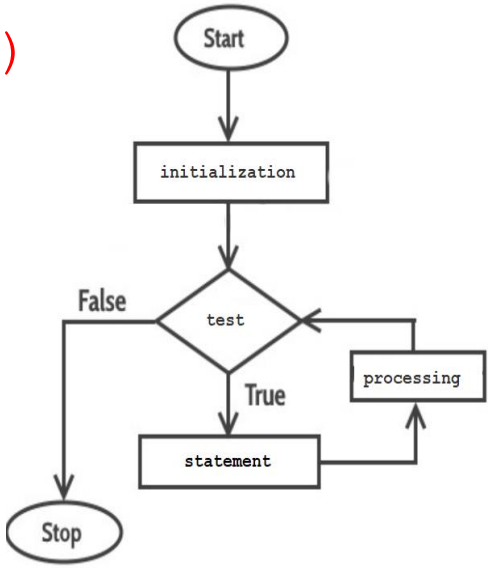
```
for (i = 1; i < 10; i+=3)
   printf("i = %d\n", i);
printf("\n\ni = %d\n", i);
```

```
for (i = 1; i < 10; i+=3)
{
   printf("i = %d\n", i);
}
printf("\n\ni = %d\n", i);
```



```
int forloopCounter = 0;
int rock = 1, paper = 45, scissors = 122, lizard= -10, Spock = 100;

for (rock = 20; paper > 3; scissors/=39, lizard++, Spock-=3, paper /=2)
{
   forloopCounter++;
}

printf("\n\n\nrock(%d)\tpaper(%d)\tscissors(%d)\tlizard (%d)\tSpock(%d)\n",
        rock, paper, scissors, lizard, Spock);
```

```
rock(20)  paper(45)  scissors(122) lizard (-10) Spock(100)
rock(20)  paper(22)  scissors(3)   lizard (-9)  Spock(97)
rock(20)  paper(11)  scissors(0)   lizard (-8)  Spock(94)
rock(20)  paper(5)   scissors(0)   lizard (-7)  Spock(91)

rock(20)  paper(2)   scissors(0)   lizard (-6)  Spock(88)
```



The `for` loop

```
int i, j;

for (i = 1; i < 3; i++)
{
     for (j = 1; j < 3; j++)
     {
          printf("%d\t", i*j);
     }
     printf("\n");
}
```

```
1   2
2   4
```

The `for` loop

```
int i, j;

for (i = 30; i < 33; i++)
{
     for (j = 35; j < 38; j++)
     {
          printf("%c\t", i+j);
     }
     printf("\n");
}
```

```
A       B       C
B       C       D
C       D       E
```

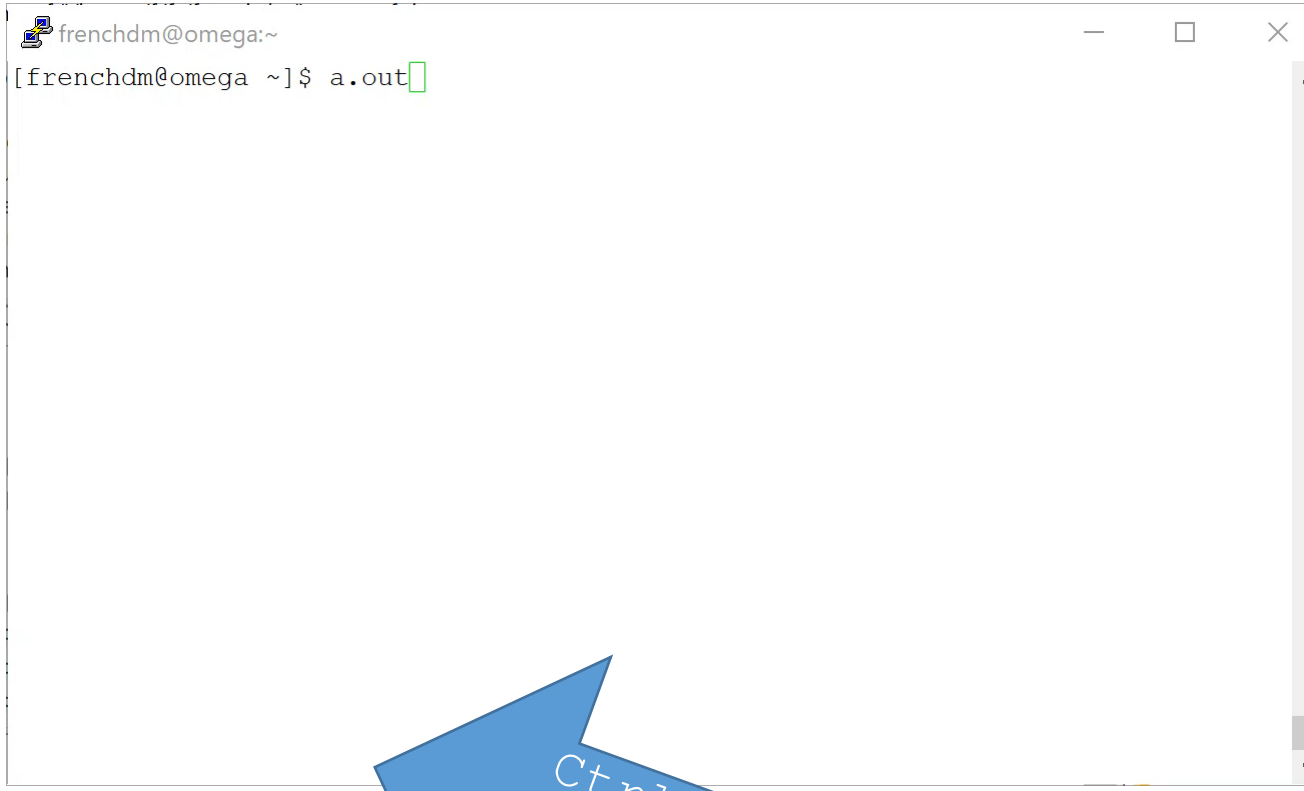
The `for` loop

```
for (i = 1; i < 4; i++)
{
      for (j = 1; j < 3; j++)
      {
            for (k = 1; k < 5; k++)
            {
                  printf("*");
            }
            printf("\n");
      }
      printf("\n");
}
```

```
****
****

****
****

****
****
```

The `for` loop

```
for (i = 1; i < 4; i++)
{
      for (j = 1; j < 3; j++)
      {
            for (k = 1; i < 5; k++)
            {
                  printf("*");
            }
            printf("\n");
      }
      printf("\n");
}
```



Ctrl-C to break out of an endless loop

The if and if-else Statements

if

- conditional statement
- allows a program to test a condition and then choose which code to execute next
- the choice depends on the outcome of that test

```
if (expression)
    statement
```

If `expression` evaluates to TRUE, then `statement` will be executed.
If `expression` evaluates to FALSE, then `statement` will not be executed.

The if and if-else Statements

if-else

- conditional statement
- allows a program to test a condition and then choose which code to execute next
- the choice depends on the outcome of that test

```
if (expression)
   statement1
else
   statement2
```

If `expression` evaluates to TRUE, then `statement1` will be executed.
If `expression` evaluates to FALSE, then `statement2` will not be executed.

```
if (DayOfWeek == SUNDAY)
{
   printf("Today is Sunda
}
if (DayOfWeek == MONDAY)
{
   printf("Today is Monda
}
if (DayOfWeek == TUESDAY)
{
   printf("Today is Tuesd
}
if (DayOfWeek == WEDNESDA
{
   printf("Today is Wedne
}
if (DayOfWeek == THURSDAY
{
   printf("Today is Thursday and tomorrow is Friday\n");
}
if (DayOfWeek == FRIDAY)
{
   printf("Today is Friday and tomorrow is Saturday\n");
}
if (DayOfWeek == SATURDAY)
{
   printf("Today is Saturday and tomorrow is Sunday\n");
}
```

```
#define SUNDAY    0
#define MONDAY    1
#define TUESDAY   2
#define WEDNESDAY 3
#define THURSDAY  4
#define FRIDAY    5
#define SATURDAY  6
```

ifDemo.c

```
if (DayOfWeek == SUNDAY)
{
        ("Today is Sunday and tomorrow is Monday\n");

        DayOfWeek == MONDAY)

        ("Today is Monday and tomorrow is Tuesday\n");

        DayOfWeek == TUESDAY)

        ("Today is Tuesday and tomorrow is Wednesday\n");

        DayOfWeek == WEDNESDAY)

        ("Today is Wednesday and tomorrow is Thursday\n");

        DayOfWeek == THURSDAY)

{
   printf("Today is Thursday and tomorrow is Friday\n");
}
else if (DayOfWeek == FRIDAY)
{
   printf("Today is Friday and tomorrow is Saturday\n");
}
else
{
   printf("Today is Saturday and tomorrow is Sunday\n");
}
```

ifelseDemo.c

Relational Operators

is less than or equal to	<=
is greater than or equal to	>=
is equal to	==
is not equal to	!=
is greater than	>
is less than	<

The actual value assigned to an expression formed with a relational operator is `1` if the relation is true and `0` if it is false.

```c
// = vs == Demo

#include <stdio.h>

int main(void)
{
    int x = 1;
    int y = 2;

    if (x == y)
    {
        printf("Hello");
    }
    else
    {
        printf("Bye");
    }

    return 0;
}
```

Is x equivalent to y?

```
[frenchdm@omega ~]$ gcc EqDemo.c
[frenchdm@omega ~]$ a.out
Bye
```

```c
// = vs == Demo

#include <stdio.h>

int main(void)
{
    int x = 1;
    int y = 2;

    if (x = y)
    {
        printf("Hello");
    }
    else
    {
        printf("Bye");
    }

    return 0;
}
```

Assign y to x and determine if x is TRUE or FALSE

```
[frenchdm@omega ~]$ gcc EqDemo.c
[frenchdm@omega ~]$ a.out
Hello
```

Operator Precedence

Relational operators have a lower precedence than any of the arithmetic operators.

```
4 <= z + 3
```

```
4 <= x = z + 3
```

```
4 <= x - z + 3
```

```
3 < x < 7
```

Relational Operators and Operator Precedence

$$3 < x < 7$$

```
#include <stdio.h>

int main(void)
{
      int x = 1;
      int x = 10;
      if (3 < x < 7)
            printf("TRUE");
      else
            printf("FALSE");

      return 0;
}
```

`(3 < x < 7)`

`((3 < x) < 7)`

What is the value of `3 < x`?

The actual value assigned to an expression formed with a relational operator is `1` if the relation is true and `0` if it is false.

So the value of `3 < x` is either 1 or 0 depending on the value of `x` which gives us either

`(0 < 7)` or `(1 < 7)` which are both TRUE

So what is the result of `(3 < x < 7)`?

Increment/Decrement Operators

++

Increment Operator

Add 1 to a variable

Two forms

```
i++
++i
```

--

Decrement Operator

Subtract 1 from a variable

Two forms

```
i--
--i
```

```
printf("1.\tThe value of counter is %d\n\n", counter);

counter = ++counter;
printf("2.\tThe value of \"counter = ++counter\" is %d\n", counter);

counter = counter++;
printf("3.\tThe value of \"counter = counter++\" is %d\n", counter);

printf("4.\tThe value of counter++ is %d\n", counter++);
printf("5.\tThe value of ++counter is %d\n", ++counter);

printf("6.\tThe value of counter-- is %d\n", counter--);
printf("7.\tThe value of --counter is %d\n", --counter);
```

counter

10

```
Enter a value for counter 10
1.      The value of counter is 10

2.      The value of "counter = ++counter" is 11
3.      The value of "counter = counter++" is 12
4.      The value of counter++ is 12
5.      The value of ++counter is 14
6.      The value of counter-- is 14
7.      The value of --counter is 12
```

Character Variables

- ASCII character set
  - 128 characters
  - each character has an integer value between 0 and 127
  - C provides an integer type named `char` to represent characters
  - `char` is stored in one byte of memory

A	65
a	97
0	48
Space	32

char1Demo.c

Character Variables

```
#include <stdio.h>

int main(void)
{
    int i;

    for (i = 33; i <= 126; i++)
    {
        printf("%d\t\tis character %c\n", i, i);
    }

    return 0;
}
```

```
33              is character !
34              is character "
35              is character #
36              is character $
37              is character %
38              is character &
39              is character '
.
.
.
120             is character x
121             is character y
122             is character z
123             is character {
124             is character |
125             is character }
126             is character ~
```

Character Variables

```
char a = 'a';
char b = 'b';
char c = 'c';
char em = '!';

printf("%d %d %d %d\n\n", a, b, c, em);
printf("a + b + c = %d\n\n", a+b+c);
printf("! + ! = %d\n\n", em + em);
printf("! + ! = %c\n\n", em + em);
printf("%c %c %c %c\n\n", a, b, c, em);
```

```
97 98 99 33

a + b + c = 294

! + ! = 66

! + ! = B

a b c !
```

getchar() and putchar()

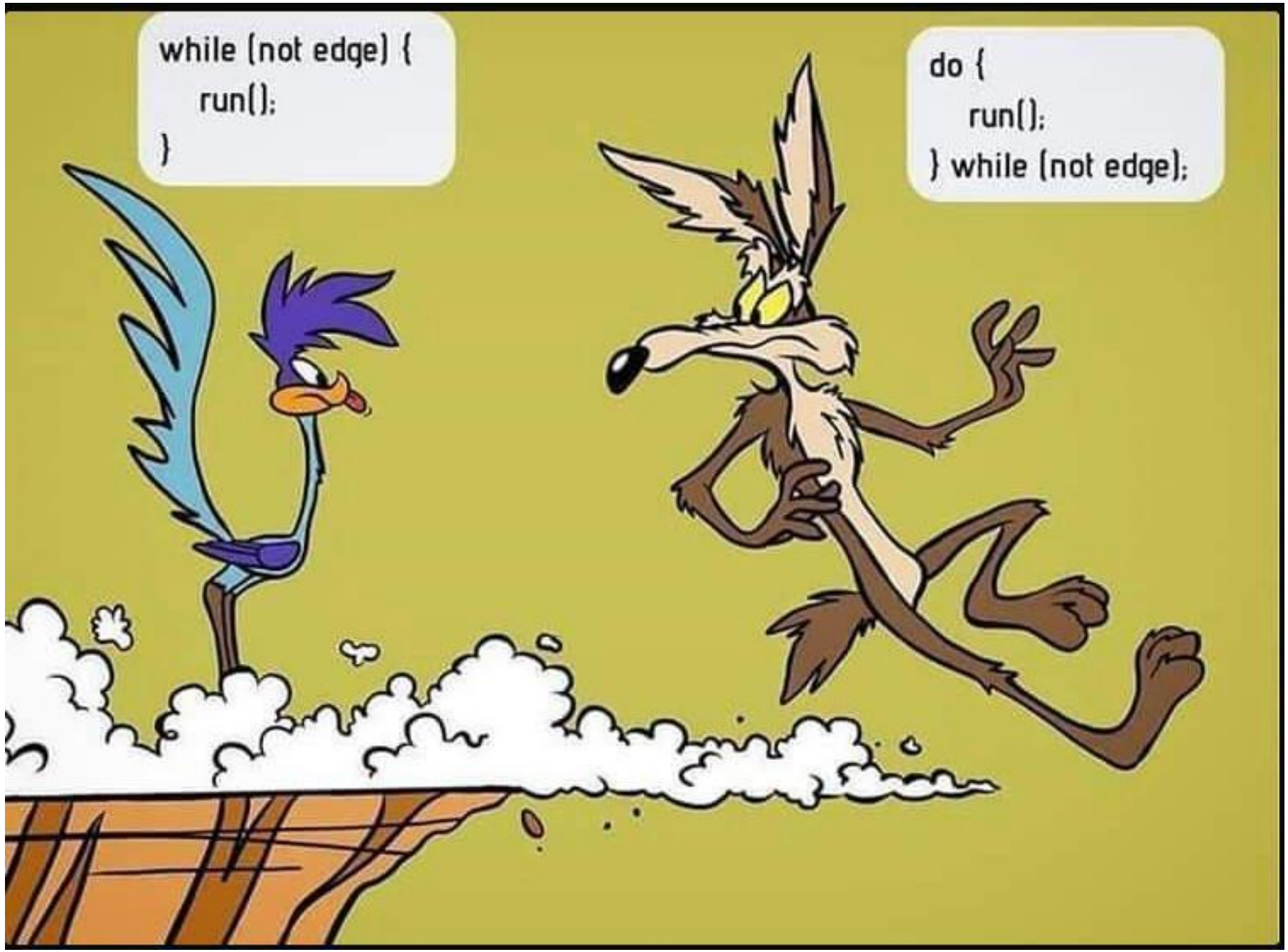
getchar()

```
int getchar(void)
```

putchar()

```
int putchar(int c)
```

```
int i;

printf("Enter a character for getchar() ");
i = getchar();
putchar(i);
```


while (not edge) {
run();
}
do {
run();
} while (not edge);

The while Loop

```
while (expression)
    statement
```

Step 1 : `expression` is evaluated

Step 2 : if `expression` is true (nonzero), then `statement` is executed

Step 3 : Return to Step 1

Iteration

executing a code segment more than once

getputwhileDemo.c

```
int main(void)
{
   int iochar;
   int LoopCounter = 0;
   int ENTERCounter = 0;
   int CharCounter = 0;

   iochar = getchar();

   while (iochar != EOF)
   {
      if (iochar == '\n')
         ENTERCounter++;
      else
         CharCounter++;

      putchar(iochar);
      iochar = getchar();
      LoopCounter++;
   }

   printf("You entered EOF - bye!!\n\n");
   printf("The while loop was executed %d times\n\n\n",
            LoopCounter);

   printf("ENTERCounter is %d\nCharCounter is %d\n\n",
           ENTERCounter, CharCounter);

   return 0;
}
```

Ctrl-D from the keyboard makes EOF

```
1Hello
1Hello
2There!
2There!
3How
3How
4are
4are
5you?
5you?
You entered EOF - bye!!

The while loop was
executed 31 times


ENTERCounter is 5
CharCounter is 26
```

CRLF vs LF vs CR

CRLF vs LF vs CR

CRLF

Windows

LF

Unix

CR

Mac

Ascii	Char	Ascii	Char	Ascii	Char	Ascii	Char
0	Null	32	Space	64	@	96	`
1	Start of heading	33	!	65	A	97	a
2	Start of text	34	"	66	B	98	b
3	End of text	35	#	67	C	99	c
4	End of transmit	36	$	68	D	100	d
5	Enquiry	37	%	69	E	101	e
6	Acknowledge	38	&	70	F	102	f
7	Audible bell	39	'	71	G	103	g
8	Backspace	40	(	72	H	104	h
9	Horizontal tab	41	)	73	I	105	i
10	Line feed	42	*	74	J	106	j
11	Vertical tab	43	+	75	K	107	k
12	Form feed	44	,	76	L	108	l
13	Carriage return	45	-	77	M	109	m
14	Shift in	46	.	78	N	110	n
15	Shift out	47	/	79	O	111	o
16	Data link escape	48	0	80	P	112	p
17	Device control 1	49	1	81	Q	113	q
18	Device control 2	50	2	82	R	114	r
19	Device control 3	51	3	83	S	115	s
20	Device control 4	52	4	84	T	116	t
21	Neg. acknowledge	53	5	85	U	117	u
22	Synchronous idle	54	6	86	V	118	v
23	End trans. block	55	7	87	W	119	w
24	Cancel	56	8	88	X	120	x
25	End of medium	57	9	89	Y	121	y
26	Substitution	58	:	90	Z	122	z
27	Escape	59	;	91	[	123	{
28	File separator	60	<	92	\	124	|
29	Group separator	61	=	93	]	125	}
30	Record separator	62	>	94	^	126	~
31	Unit separator	63	?	95	_	127	Forward del.

CRLF vs LF vs CR

```
cat file.txt | tr '\r' '\n' | tr -s '\n' > file.translated.txt
```

This Unix command will translate the
CR in Mac files or the CRLF in Windows files to
UNIX LF

You can also use

```
sed -i.old 's/\r$//' MyFile.txt
```

Logical Operators and Expressions

logical-not	!
logical-and	&&
logical-or	||

```
!expression1

expression1 && expression2

expression1 || expression2
```

Logical Operators and Expressions

logical-not	!
logical-and	&&
logical-or	||

p	q	!p	!q	p && q	p || q	!(p && q)	!(p || q)	!p && !q	!p || !q
1	1	0	0	1	1	0	0	0	0
1	0	0	1	0	1	1	0	0	1
0	1	1	0	0	1	1	0	0	1
0	0	1	1	0	0	1	1	1	1

Precedence of the Logical Operators

Logical-not (!) has higher precedence than the logical-and (&&) which has higher precedence than the logical-or (||)

- logical-not
  - logical-and
    - logical-or

- Left to right evaluation

```
i || !j || !k          (i || (!j)) || (!k)
i && !j && !k          (i && (!j)) && (!k)
i || !j && !k           i || ((!j) && (!k))
```

Logical Operators and Expressions

Caution

C only evaluates as much as necessary to determine the truth value.

This is called "**Short Circuit Logic**"

&&

The second operand will only be evaluated when the first operator is nonzero.

||

If the first operand is nonzero, then the second operand is not evaluated.

logicalevaluationDemo.c

```
int main(void)
{
   int i = 0;
   int j = 0;

   printf("i = %d   j = %d\n\n", i, j);
   printf("i && j++ evaluates to %d\n\n", i && j++);
   printf("i = %d   j = %d\n\n", i, j);
   printf("i || j++ evaluates to %d\n\n", i || j++);
   printf("i = %d   j = %d\n\n", i, j);

   printf("Resetting i and j to 0...\n\n");
   i = j = 0;

   printf("i = %d   j = %d\n\n", i, j);
   printf("i && ++j evaluates to %d\n\n", i && ++j);
   printf("i = %d   j = %d\n\n", i, j);
   printf("i || ++j evaluates to %d\n\n", i || ++j);
   printf("i = %d   j = %d\n\n", i, j);
   return 0;
}
```

```
i = 0   j = 0

i && j++ evaluates to 0

i = 0   j = 0

i || j++ evaluates to 0

i = 0   j = 1

Resetting i and j to 0...

i = 0   j = 0

i && ++j evaluates to 0

i = 0   j = 0

i || ++j evaluates to 1

i = 0   j = 1
```

XOR

- Only TRUE if one or the other is true but not both.
- One or the other but not both.

p	q	p XOR q
T	T	F
T	F	T
F	T	T
F	F	F

You can have ice cream or pie for dessert but not both.

You can go to sleep at 8AM or go to work at 8AM but not both.

XOR

```
1 && 1        TRUE        1 && 0        FALSE
1 || 1        TRUE        1 || 0        TRUE
1 ^  1        FALSE       1 ^  0        TRUE

0 && 0        FALSE       0 && 1        FALSE
0 || 0        FALSE       0 || 1        TRUE
0 ^  0        FALSE       0 ^  1        TRUE
```