*"Mr Scoot, why do all the Red Alert Alarms keep going off ???"*

--- James T. Kurt
Captain, USS Enterprise

# Looping

The **factorial** of an integer N, written N!, is the product of consecutive integers 1 through N. For example, 5! is calculated as follows:

5! = 5 * 4 * 3 * 2 * 1

or

120

The following is a program one could write to calculate the **first 5 factorial numbers**. You can watch it work at: http://ideone.com/oNUBb:

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

    int factorial; /* current factorial value */

    factorial = 1;  /* first factorial */
    printf ("1! = %i \n", factorial);

    /* second factorial */
    factorial = factorial * 2;
    printf ("2! = %i \n", factorial);

    /* third factorial */
    factorial = factorial * 3;
    printf ("3! = %i \n", factorial);

    /* fourth factorial */
    factorial = factorial * 4;
    printf ("4! = %i \n", factorial);

    /* fifth factorial */
    factorial = factorial * 5;
    printf("5! = %i \n", factorial);

    return (0);
    }
```

**Output:**

```
1! = 1
2! = 2
3!  = 6
4! = 24
5! = 120
```

---

This will work for small numbers, but what if you have 100, 1000, or a 1,000,000 numbers to calculate. You would be correct in assuming that the program would be very large and not maintainable.

**Looping** is a construct in most languages that allows statements to be repetitively executed. It enables a programmer to develop concise programs containing repetitive processes that could require 1000's or millions of statements to be performed.

C has **three types** of looping mechanisms: **for, while and do.**

---

A word about a **code optimization** issue in the program above.  To optimize this code a bit better, consider the following statement:

```
factorial = factorial * 5;
```

Its always better to use the binary assignment operator for multiplication instead ( * = ) as it will generate less executable code that will be run.

```
factorial *= 5;   /* much faster */
```

Its little things like this over the long run that can really improve the speed and performance of your code when it executes.   Especially when statements like this might be executed multiple times within the loops we will be discussing this week.

# Star Trekking - The Red Alert Loops

In this week's episode, it seems they are having trouble on the Enterprise with the "Red Alerts" going off every other minute.  Captain Kurt is quite irritated by this experience, and looks to chief engineer Mr. Scoot ("Scotty") to immediately fix the issue.  It appears that the "looping code" that implements the security system has some issues.

# Video - C Looping, The Operators

This is a good video that provides some animation on how various operators work with loops in C (or really any other language). This will help you set up the right tests and conditions to make everything work correctly.

# Relational Operators

Like other languages, C has **relational operators** which return one of two values, TRUE or FALSE. A value of **FALSE** in interpreted by the C compiler as a **value of 0**, while **TRUE** is interpreted as a **value of 1**. Note that whether the result is floating point (0.0) or integer (0), that both are zero and both values are **FALSE**. If you want to take it a step further, **zero is FALSE**, any other value would be **TRUE**. The values -1, 0.3, 34 are all **TRUE** values, only 0 and 0.0 are **FALSE**. You might be asking what about **characters** like 'A', 'b', '\n', and others ... rest assured, we will cover them in depth in Chapter 10 with our discussion on character strings, but for now, every character in C is actually converted to an integer value, and the only character that has a zero integer value is the null character ('\0').

If you decide to **print** the value returned from some relational expression, it will only print a value of 1 for TRUE and 0 for FALSE. If you use a format such as floating point, it would print 1.0 or 0.0, but that's just printing semantics as the expression returns just 1 or 0.

Shown below are the common relational operators, with the exception of the first two ( = = and ! = ), the others are so common that we all leaned about them in elementary school.

| | |
|---|---|
| **= =** | **EQUAL TO** |
| **!=** | **NOT EQUAL TO** |
| **<** | **LESS THAN** |
| **<=** | **LESS THAN OR EQUAL TO** |
| **>** | **GREATER THAN** |
| **>=** | **GREATER THAN OR EQUAL TO** |

If you learn one thing this week, this is the one item you really need to understand as it can get you into a lot of trouble if misused in C:

## = = IS NOT THE SAME AS =

The **is equal to** operator ( = = ) is a relationship. when you say

```
val == 2
```

You are asking the question whether val **is equal to** 2, absolutely no variable or memory location is updated when this statement is executed. If the contents of val are indeed 2, then a TRUE value is returned, otherwise it is FALSE. The value of val is **unchanged**. The exact opposite of the **is equal to** operator ( **= =** ) is the **not equal to** ( **! =** ) operator. It just gives the ability to verify if a value *is equal to* some value, or *is not equal to* some value.

The use of an **=** operator implies an **assignment**, ... when you say:

```
val = 2;
```

... it **stores** the value of 2 into the variable called **val**:

**val**

$\boxed{2}$

---

## Precedence with Relational Operators

The relational operators have ***low precedence***. When you have the following:

```
a < b + c;
```

The addition is done first and then the result of b + c is compared against a: a < (b + c);

If a < b + c is FALSE, the ***result*** could be assigned to a variable and would be zero.

```
x = (a < b + c);  /* x would be assigned either 0 or 1 */

printf ("x = %i \n", x);   /* could only print 0 or 1 */

printf ("%i", a < b + c);  /* could only print 0 or 1 */
```

Reference **Appendix A** in the Kochan book for a table that shows **operator precedence in C**. If you don't have a copy on hand, check out this nice link complements of **Swanson Technologies**, which like most things, is copyrighted, so I'll provide proper credit and just link to it:

http://www.swansontec.com/sopc.html

# Pre and Post Operations

C has operators that allow for a variable to **increment** or **decrement** a value by 1, but the time this happens depends on how the operator is used. The operator **++** will add a value of 1 to a variable, while **- -** (two minus signs) will subtract it by one. The key is ***where*** these operators appear in a statement as it will determine if it executes in the current statement or immediately following the statement.

Let's look at two examples that show the difference between a **pre increment** and a **post increment**.

## Post Increment Example

In the example below, the ++ operator is shown AFTER the number, which identifies it to the compiler as a **post increment** operator, meaning that n will be incremented by one after the print statement is executed.

Given:

```
n = 5;

printf ("%i \n", n++); /* post increment */
```

would be treated by the compiler as:

```
n = 5;

printf ("%i", n); /* prints 5 first */

n = n + 1; /* then increments */
```

## Pre Increment Example

Let's contrast that with a **pre increment** operator in the code below that will increment the number first, then execute the print statement. Note that the ++ operator is BEFORE the variable.

```
k = 5;

printf ("%i \n", ++k);  /* pre increment */
```

would be treated by the compiler as:

```
k = 5;

k = k + 1;  /* increments first */
```

```
printf ("%i \n", k);  /* then prints 6 */
```

In both cases, the pre and post increment operators set their variable values to 6, its just a matter of when that happens and it DOES make a difference *most* of the time.

For example, it *does not* make a difference in this case since k and n are both simply implemented with nothing else going on:

```
k = 5;

n = 5;

++k             /* k is now 6 */

n++             /* n is also now 6 */
```

## Final Thoughts

The **pre and post decrement** operators work the same way, it is just that they subtract a number by one instead of adding it by one.   Just substitute the - - operator in place of the ++ operator and try out the statements above.

Novice C programmers have a difficult time picking up these operators initially and often get them confused.    The important thing is that they SHOULD be used as they are **faster** than just adding or subtracting a value by one.   To illustrate the difference of adding a number by one to a variable, let's look at two statements:

```
value = values + 1;
```

**verses**

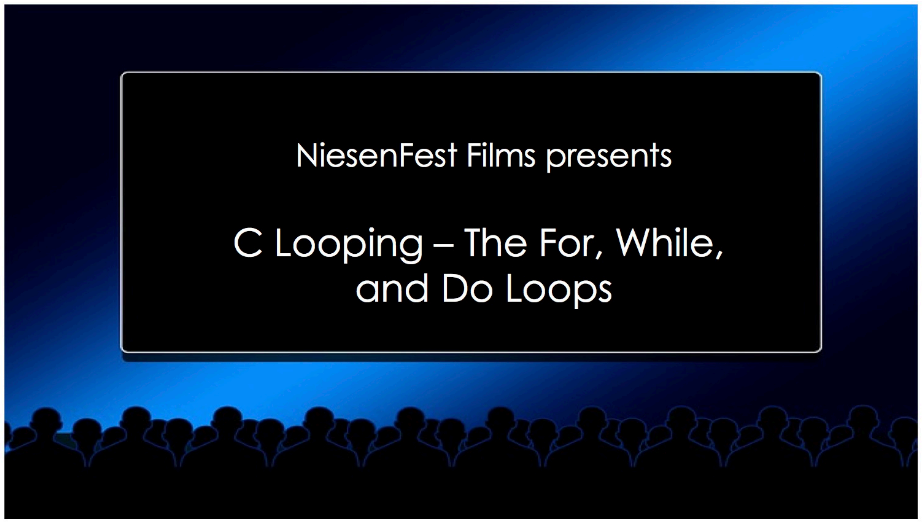
```
++value;
```

Both statements will add one to the variable value.   The **BIG difference** is that the first statement will likely generate ***more assembly/machine code*** to execute it, which in turn will make it ***slightly slower*** as it has to execute more code.   Remember that C is a high level language that is translated into assembly/machine code, and that is the code that is run on your computer.

Now many of you are saying:  "Tim, why would that make such a difference?".   If you have just a couple of statements like this in your code, probably not much, but what if those statements are located in a ***function*** or ***loop*** that is ***executed many times*** in your program?   In the end, they all add up, and this is just one example of a simple thing you could do to help speed up your program.   In the real world, microseconds or seconds can mean the difference between a missile hitting its target, a car's computer system sensing when to apply anti-locking brakes, or you just sitting and waiting for your ATM transaction at the bank to complete.   I'll be showing you the little things you can do to make sure your programs are executing as fast as possible, but with a balance of making sure they are easily testable and readable, as you don't want to make things so complex for the sake of speed.

## Video - C Looping, The Loops

This video concentrates on the three actual loops that are part of the C Programming Language. You will see extensive animation to help you better understand what each line of code in C is doing, each step of the way in the loop, in terms of memory and input/output.

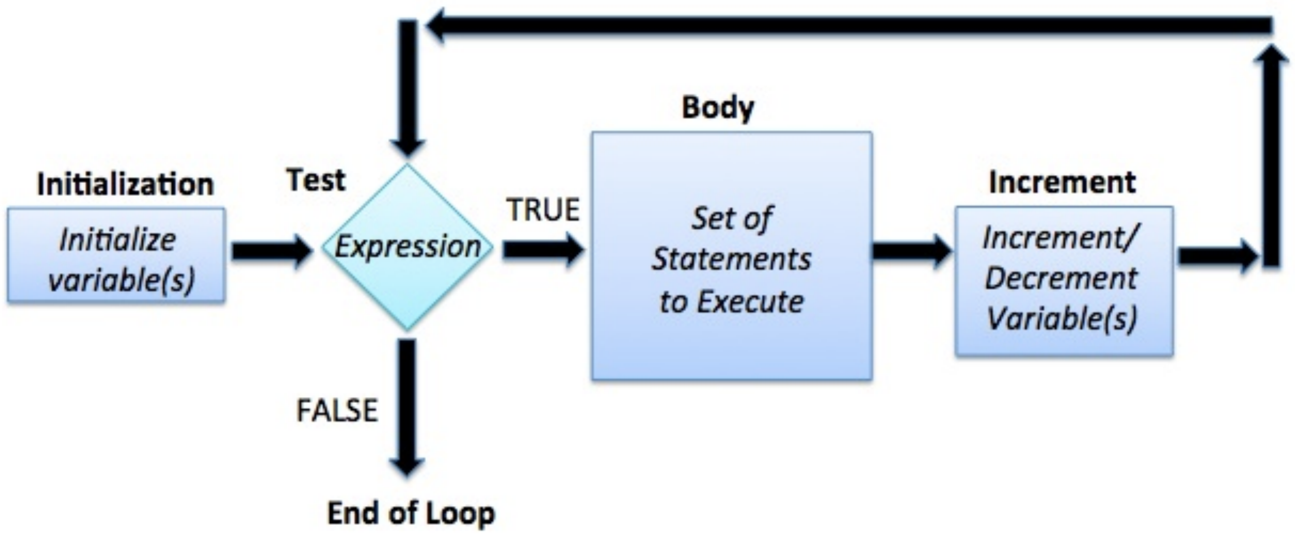

NiesenFest Films presents
C Looping – The For, While, and Do Loops

# For Statement

A **for statement** is used in C to repetitively execute a series of statements within its **body**. In my class, I will refer to a **body** as 0 or more statements. A **statement** is typically any C code segment that ends with a semicolon, such as:

```
avg = hits / at_bats;
```

Below is a useful diagram of the three major **steps** in a "for loop", which includes **initialization**, **test**, and **increment**.



Here is an example using a ***for loop*** that **increments** the variable **idx** from **1** to **5**. Each time through the loop, the variable **idx** will be *incremented* by 1.

```
for ( idx = 1; idx <= 5; ++idx )
{
     printf ( "...%i", idx );
}
```

It would print the following:

```
...1...2...3...4...5
```

You can also **decrement** a variable in a ***for loop***. Here is an example of a for loop that counts from **5** to **1**. Each time through the loop, **idx** will be *subtracted* by 1.

```
for ( idx = 5; idx >= 1; - - idx )
{
     printf ( "...%i", idx );
}
```

It would print the following:

```
...5...4...3...2...1
```

## How to prompt for values to use with a Loop

The complete program example below shows how you can have your code **prompt** to enter the number of times you want a loop to execute. The value entered as input to the program is stored in an integer variable called **loop_count**. The two ***for loops*** then use the integer ***value*** stored in the variable **loop_count** to execute each loop shown below. Note how the value **5** is entered as **input** to the program in the IDEOne **stdin** area and what output is generated when you watch it live at: http://ideone.com/WQwSJ6

```
/* This program will prompt for the number of times */
/* a loop will be executed.  It shows how to print */
/* a loop going from 1 to the loop count and */
/* vice versa from the loop_count to 1. */

#include <stdio.h>
int main ()
{
   int loop_count;  /* number of times to loop */
   int idx;               /* loop index */

  /* Prompt for the number of times to loop */
   printf ("Enter the number of times to loop \n");
   scanf ("%i", &loop_count);

   printf ("\nIncrementing; \n");
   for ( idx = 1; idx <= loop_count; ++idx )
   {
      printf ( "...%i", idx );
   }

   printf ("\n\nDecrementing:\n");
   for ( idx = loop_count; idx >= 1; --idx )
   {
      printf ( "...%i", idx );
   }

   return (0);

}
```

The ***input*** to this program is 5, but it can be any reasonable *integer* value (i.e., don't put a huge number in like a million or billion as your program would start printing lots of numbers on the screen and would likely time out or take a long time to complete). If you are running with IDEOne, the input values are entered in the **stdin** area of the IDEOne screen, representing the input that would be passed to the program. If you are using another compiler that is installed on your computer such as Visual Studio, you would be prompted on the screen to enter an integer value, in this case, you would simply enter a 5 and hit return.

```
5
```

The ***output*** to the program based on the input value of 5 would be:

```
Enter the number of times to loop

Incrementing:
...1...2...3...4...5

Decrementing:
...5...4...3...2...1
```

## A Detailed Look at the For Loop

Here's a recap of how a for loop works:

```
for (INTIALIZATION; TEST; INCREMENT)
      BODY;
```

1. **STEP 1:** INITIALIZATION is evaluated first, done only once in the beginning.
2. **STEP 2:** If the TEST is not satisfied (i.e., False value), the loop is immediately terminated. Execution continues with the first statement following the for loop.
3. **STEP 2a:** Program statements which constitute the body are executed.
4. **STEP 3:** Values specified in the INCREMENT/DECREMENT section are executed.
5. **STEP 3a:** Return to Step 2

Note the following about the for statement:

- **Initialization** is only done once
- The **test** is done before the **body** is executed
- If the **test** is false, the **body** is not executed, thus the **body** may not be executed at all
- The **body** can be multiple statements in braces

With the last point, let's look at this example:

```
for (i = 1; i < = 10; i++)
  printf ("In Loop\n");
  printf ("Not in Loop\n");
```

The above would really be seen by the compiler as:

```
for (i = 1; i < = 10; i++)
{
   printf ("In Loop\n");
}
printf ("Not in Loop\n");
```

**TIP:** It's always a good idea to use the braces ... that way if you add statements later, you won't fall into the trap of those statements not being part of the loop. Braces *do not add any executable code* to your program.

**Naming Convention** - It's OK to use very simple variable names like **i** or **j** for loop indexes ... no need to spell it out, such as index. The for loop can have many forms. Here are just a few:

```
for ( n = 1, j = 1; j < 10; ++n, ++j )  /* Multiple initializations and increments */

for ( ; ; )  /* infinite loop, no test to exit this loop  */
   printf ("All work and no play makes Jack a dull boy");  /* see the movie - The Shining */

for (; j < 10; ++j)     /* no initialization, but                */
   body;             /* j should be initialized/set somewhere */
                     /* before the for loop                   */

for ( j=1; j < 10 ; )   /* no increment step */
{
    x += 5;
    printf ("%i \n", x);
    j += 3;   /* increment step done in the body */
 }
```

If a part of the for statement is left out, the semicolon must be present. The body of a for loop can contain another **nested for** statement.

```
for (i = 1; i < 10; ++i)
{
    for (j = 1; j < 20; ++j)
   {
         printf (" %i  %i \n", i, j);
   }
 }
```

The above statement could also be written as the statement below, but I would recommend the previous example for readability and maintainability.

```
for (i = 1; i < 10; ++i)
   for (j = 1; j < 20; ++j)
     printf (" %i  %i \n", i, j);
```

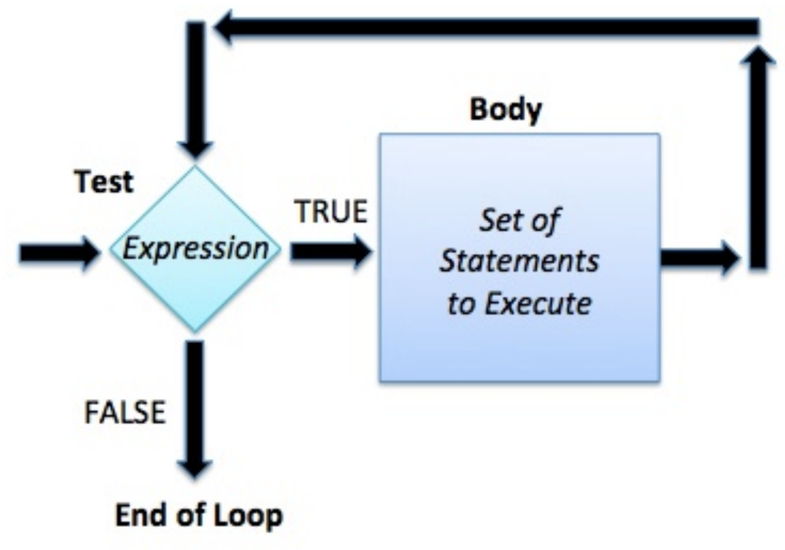
* Remember that braces are needed if you wanted to add more statements to either of the BODY's of the for loops in this example.

* It does not hurt to add braces with loops, even if you have only one statement in the body, and is actually a recommended practice in Industry.

# while Statement

The **while loop** is a very popular loop mechanism that is found in most languages. Unlike a for loop, it requires you to do your own **initialization** and **incrementing** if needed in order for the loop to work correctly. It has the following syntax:

```
while (test)
   BODY;
```



The following example shows **initialization** prior to the loop, the **test** in the while statement, and the **incrementing** inside the body of the loop.

```
idx = 1;  /* initialization */
while (idx <= 5)
{
     printf ("...%i", idx);
     ++idx; /* increment, same as:  idx = idx + 1; */
}
```

It would print: ...1...2...3...4...5

Here is another example of a while loop that **decrements** each time inside the loop. If you want to watch both of these loops in action, watch them run in IDEOne at: http://ideone.com/nCj15F

```
idx = 5;  /* initialization */
while (idx >= 1)
{
     printf ("...%i", idx);
     - - idx; /* decrement, same as:  idx = idx - 1; */
}
```

It would print: ...5...4...3...2...1

Compare to for statement:

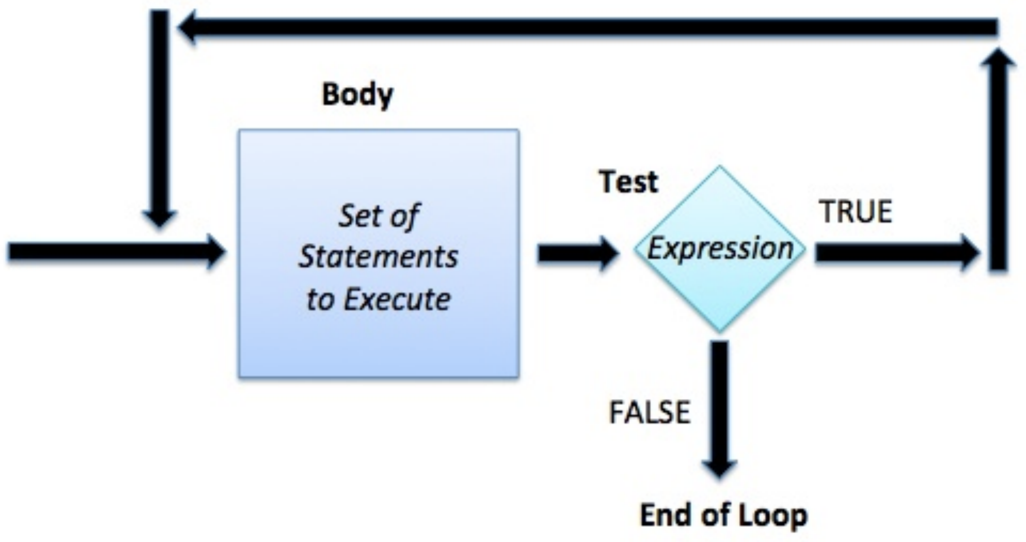
- **NO INITIALIZATION** ... If needed, you must do your own initialization, using the assignment operator.
- **NO INCREMENTING or DECREMENTING** ... If needed, you must do your own incrementing/decrementing, using arithmetic operators.
- Same as **for (; test ;)** ... if test is false the first time, the body will not be executed.

## do while statement

The **do while loop** is great when you know at a minimum that you will have to process your loop body of statements at least once. An example might be reading a line in a file and processing it. At a minimum you have to read the first line in the file. If the file is empty, then the file will contain a value that indicates EOF - or end of file. We'll learn more about this in Chapter 16 (Input and Output), but there is definitely a place for this loop when creating C programs. The **do loop** is a looping mechanism that has the following syntax:

```
do
{
   body;   /* set of statements */

} while (test);
```



Example of a **do while** loop that **increments** from 1 to 5

```
idx = 1; /* do your own initialization */

do {

     printf ("...%i", idx);
     ++idx; /* do your own increment */
} while ( idx <= 5 );
```

It would print: ...1...2...3...4...5

Here is the same example, but **decrementing** from 5 to 1. You can watch both of these do loops run at: http://ideone.com/ZAgxJH

```
idx = 5;  /* do your own initialization */
do  {
    printf ("...%i", idx);
    - - idx;  /* do your own decrement */
 }  while ( idx >= 1 );
```

It would print: ...5...4...3...2...1

---

Compare the **do loop** to a **for** and **while** loop:

- **NO INITIALIZATION, NO INCREMENTING** as part of the **do** statement.
- The **body *will always be done at least ONCE***, since the **test** comes after the body

# C99 Variable Declarations and Looping

The **C99 standard** for the C language added the ability to declare variables inside constructs such as loops, conditionals, and other items we have yet to cover. The benefit of declaring variables inside of the loop is that access and scope is limited only within that loop. These variables can't be corrupted later on in other parts of the program. You can even reuse the variable names in other loops, but it won't remember any of their previous values.

Let's consider the following program. Note how the variables **idx** and **hours** are declared within the for loop, while the variable **sum** is declared in the beginning of the main function. I've highlighted the three declarations in bold so that they stand out for you to review. You can try it out at: http://ideone.com/BefJoN

```
#include <stdio.h>
int main ()
{
   float sum = 0;  /* sum of the employee hours, not using C99 */

   for (int idx = 1; idx < 5; ++idx) {   /* idx is a loop index, C99 */
      float hours; /* hours worked in a given week, C99 */

      printf("Enter Hours Worked: ");
      scanf ("%f", &hours);
      sum += hours; /* calculate sum of hours as a running total */
   } /* end for */

   printf ("\n Total sum of hours is %6.2f \n", sum);

   /* if you try to reference the variables idx or hours here, you will get a
syntax error */

   return (0);

} /* end main */
```

At this point, the variables **idx** and hours are declared within the loop and once you leave that loop you can't reference them anywhere else in the main function. For me, declaring the loop index **idx** inside the for statement is a no-brainer if you want to code with security and performance in mind. As for the **hours** variable, if you never intend to use it again in your program, then feel free to declare it within the loop. The main benefit of this approach is that the **scope** of the variables **idx** and **hours** are limited to the loop, and you don't have to worry about their values getting corrupted later on in the program.

The variable **sum** was not declared in the loop, and thus can be **used anywhere** in the main function, and you can see in our program that it is used INSIDE the loop and in the code AFTER the lo*op*.

The purpose this lecture note is to make you aware of the benefits of this standard. I won't take off points on any homework assignment or exam question if you never utilize this particular feature of C99, but I wanted you to be aware of it and to consider it going forward in this class or in future classes. Try out the example above in IDEOne and experiment a bit with it.

## break statement

The **break command** implies an immediate exit from the loop. Just like a "Go directly to Jail" card in Monopoly: "Go directly to jail, do not pass GO." Execution will continue with the first statement following the loop. Let's look at an example that you can also try out at: http://ideone.com/Ypx6If

```
#include <stdio.h>
int main ()
{
   int a = 10;  /* just a variable */
   int i;           /* loop increment */

   for (i = 1; i < 10; i = i + 2)
   {
      a += 5;
      break;
      a = a + 100;   /* this statement never executed */
   }

   /* first executable statement after the loop */
   printf ("after the loop \n");

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

   return (0);

} /* end main */
```

---

The final value of **a** is 15. Because of the break, the statement **a = a + 100** is not executed. The loop ends.

The final value of **i** is 1. The contents of **i** are not incremented because of the **break** statement.

The break statement can be used inside of any type of loop

In reality, this is a stupid example, because **a = a + 100** can never be executed. It makes more sense for it to be part of a conditional statement (such as an if, which we will learn more of in Chapter 6). If you compile this, you will probably get a warning that that statement could never be executed. You can still run it since its not a compiler error, but just a warning.

Personally, I normally avoid the use of the break statement when used in a loop.   I'll show you next week how to set flags and use conditional and Boolean logic to do this better.

# continue statement

The **continue statement** continues execution but skips all statements between continue and the end of the loop. I rarely use a continue statement, but show it here just so you know how it works. Like an exit or break statement, you would normally use this with an if statement that we will cover next week. Watch it work in IDEOne at: http://ideone.com/DLpH45

```
#include <stdio.h>
int main ()
{
   int a = 10;  /* just a variable, start at 10 */
   int i;          /* loop increment */

   for (i = 1; i < 10; i = i + 2)
   {

      a += 5;
      printf ("In Loop:  i = %i, a = %i \n", i, a);   /* print final values */
      continue;
      a += 100;   /* ignore this statement, it will never be executed */

   }  /* end of the loop */

   printf ("after the loop\n");   /* first executable statement after the loop */
   printf ("Final Values:  i = %i, a = %i \n", i, a);    /* print final values */

   return (0);

} /* end main */
```

What is the value of a? or i? Let's look at the output of this program.

```
In Loop:  i = 1, a = 15
In Loop:  i = 3, a = 20
In Loop:  i = 5, a = 25
In Loop:  i = 7, a = 30
In Loop:  i = 9, a = 35
after the loop
Final Values:  i = 11, a = 35
```

# exit statement

The **exit statement** immediately exits the program. It should be used with care and it not only exits the loop in this case, but it ***exits your program***! It has the following syntax:

```
exit (exit_status); /* 0 is a good status, non-zero for abnormal exits */
```

Here is an example:

```
#include <stdio.h>
#include <stdlib.h> /* need to include stdlib for exit to work correctly */
int main ()
{
   int a = 10; /* just a variable */
   int i; /* loop increment */

   for (i = 1; i < 10; i = i + 2)
   {
      a = a + 5;
      exit (1);  /* immediately exit the program */
      a = a + 100;  /* this statement NEVER gets executed */
   }

   printf ("after the loop\n"); /* first executable statement after the loop */
   return (0);

} /* end main */
```

What is the value of **a**? or **i**?

The final value of **a** is 15.

The final value of **i** is 1.

---

You can exit from different sections of your program, and give a different value in the call as a parameter (like **exit (2);** ). That way, it is possible to tell when execution prematurely exits from a program. In many cases, due to some error condition. An example would be if you were unable to access a needed file or database in order to process a list of employees for payroll. I like to end my main function with a **return (0);** as it lets me know that it completed normally.

Note that a = a + 5; is better written as **a +=5;** ... same for a = a + 100 ... **a += 100;**

On a final note, one common error I see later on in the class is that students confuse an exit statement with a return statement. A return statement is commonly used with functions to return back to the calling function (we'll discuss this in a few weeks). An exit statement will immediately **abort** your program.

# Common Looping Errors

As you start using loops in C, keep these issues in mind as they are common mistakes made by new, and sadly, experienced C programmers. In most cases, the C compiler will not generate an error at compile time, but you'll notice your program will not execute correctly at run time.

## 1) Putting a semicolon after the "while" or "for" loop statement

Doing this will not generate a compiler error, but will cause all the statements in your loop body to not be executed at all, or at most, just one time. You can easily do this with both a while and for loop. The key point here is don't add a semicolon at the end of the **for** or **while** statement.

```
for (i=0; i < 10; ++i) ;  /* bad semicolon */
{

     /* for loop body */
    printf ("Hello, is anyone there? \n");
    printf ("The loop may only execute at most one time \n");

} /* for loop */
```

Note that for a **do while** loop, there is a semicolon after the while part of it, but also note it is **after** the loop body.

## 2) Loop off by one error

This is a very common error where the loop execution is off by 1, either one less or one more. For example, what if you wanted to print a count from 1 to 10:

```
for (i=1; i < 10; ++i)
{
    printf ("%i …\n");
}
```

… this will stop when it gets to 10, and 10 would not be printed. To fix, use a **less than or equal** test (**< =**) instead of less than (**<**) in your loop test. This is probably the most common loop issue that plagues both novice and experienced programmers.

```
for (i=1; i < = 10; ++i)
{
    printf ("%i …\n");
}
```

## 3) An undefined or un-initialized value used in the loop test

Here is a case where i is **never initialized** to any value. Remember that in C, you can't assume that values are initialized to 0.

```
int val;                    /* val is undefined at this point */

for (; val <= 10; ++val)    /* val is still undefined here, could be anything!  */
```

Can be **fixed** in two ways:

a) Initialize val when you declare it

```
int val = 1;
```

b) Initialize val in the for loop initialization section

```
for (val = 1; val <= 10; ++val)
```

## 4) Infinite Loop

An infinite loop runs **forever**. The C run time environment will eventually notice this and processing will be terminated at some point. In many cases, it is not always obvious, here are a few examples:

```
for (; ; )  /* obvious, no loop test here, no initialization or increment either */
{

   /* recognize what Movie this came from? */
   printf ("All work and no play makes Jack a dull boy \n");
}

for (i=0 ; ; ++i)                /* no loop test , just an initialization and
an increment step */

for (val = 10; val > 0; ++val)       /* val will always be greater than zero */
```

It is important to note that in some cases, an infinite loop is exactly what is needed, especially if you are kicking off and processing various processes and threads within real time programming (http://en.wikipedia.org/wiki/Real-time_computing).

However, in this class, an infinite loop would never be appropriate for any code that we'll be doing this semester. If for some reason your program does not stop running, on many operating systems, such as UNIX, you can stop and kill your process that is running, ... and a control key along with a "c" or "z" key will sometimes work.

# Sample Exercises to Try Out (Looping Chapter)

Below are some sample exercises to try out that are based on what we learned this week about loops in C. They are taken directly out of the Kochan book in case you do not happen to have that book. These exercises are NOT HOMEWORK and DO NOT need to be sent to me ... I will provide the answers to each one on next week's lecture notes page. Try to see if you can do them on your own first, and then compare your answers with my suggested answers next week. The important thing here is that you gain valuable experience in solving problems with the C Programming language.

**Looping Chapter:  Sample Exercises from the Kochan Book**

**Sample Exercise 1**

*/* Sample Exercise 1*

*Write a program to generate and display a table of n and n squared, for integer values ranging from 1 to 10.  Be sure to print appropriate headings*

*Hint:  Good use of a loop ... C has no operator for square ... so use* ***num * num i****n the code*

**/*

**Sample Exercise 2**

*/* Sample Exercise 2*

*A triangular number can also be generated by the formula*

*Triangular number = n (n + 1) / 2*

*for any integer value n.  Write a program to generate every fifth triangular number between 5 and 50 (5, 10, 15 ... 50)*

*Hint:  Good place to use a loop ... increment each time  by 5*

**/*

**Sample Exercise 3**

*/* Sample Exercise 3*

*Given the a factorial is the product of consecutive integers 1 through n, write a program to calculate the first 10 factorial values*

*2 factorial = 1 * 2*

*3 factorial = 1 * 2 * 3*

*4 factorial = 1 * 2 * 3 * 4*

*Hint:  Perfect place to use any loop ...*

**/*

**Sample Exercise 4**

*/* Exercise 11*

*Write a program that calculates the sum of the digits of an integer.  For example, the sum of the digits 2155 is*

*2 + 1 + 5 + 5 or 13.  This program should accept any arbitrary integer typed in by the user.*

*Hint:  To strip out digits, look into using the mod function ... divide numbers by 10*

*345 % 10 ... the remainder is 5, which happens to the right most digit ...*

*... if you divide the number by 10, you get 34 ... the remaining digits*

*... if you put this idea in a loop, you could extract the 3, 4, and 5 digits*

**/*

## Summary

In our first set of notes this week, we covered the basics of how a C program works with variables of various data types such as integers, floating point, and characters ... as well as the numeric systems supported (decimal, octal, and hexadecimal). Additionally, we reviewed how to read and display each variable type and showed how to work with many general purpose arithmetic operations (add, subtract, multiple, division, and modulus) along with various conversion rules based on transition to another type.

A second important programming concept was presented in the other set of notes this week. Looping is a construct in most languages that allows statements to be repetitively executed. It enables a programmer to develop concise programs containing repetitive processes. We covered three basic loop constructs: a for, while, and do while. We also reviewed statements and system calls that handle control and out of a loop: exit, continue, and break.

## For Next Week

For next week, please read the next chapter on *Conditional* processing. If you have time, try some of the exercises from the chapters we covered this week, and I'll post a few of my answers in next week's lecture notes.  Recall that selected answers to textbook questions by the author can be found in the **Resources** folder on our class page.

As with most weeks going forward, a **Quiz** and **Homework Assignment** will be due next Sunday at midnight.  Please see the Quiz and Homework Assignment links at the end of the Looping notes.   You can work on the Quiz and Assignment anytime during the week, but remember to finish both by their due dates.  Additional details and hints on  your first assignment are attached to the assignment link.   Make sure you that you read and follow the class **coding standards** that were made available on our class home page this week.

For you **Star Trekking** fans out there, next week the crew is on a mission to deliver urgent medical supplies to Voltran 5.   In order to do that, they must cross the "Neutral Zone" which is guarded by those "pesky" aliens known as **Klingons**.  They impose many **conditions** on Captain Kurt,  who has to weigh the cost and benefits of each possible **decision**.   This type of "thinking" just happens to be the type of logic in C that we will study next week.

# HOMEWORK 1 - PROGRAM LOOPING

Write a C program that will calculate the pay for employees.

For each employee the program should prompt the user to enter the clock number, wage rate, and number of hours. This input will be used to determine the gross pay for a given week, which is the number of hours worked in a given week multiplied by the employee's hourly wage.

Use the following data as test input for clock number, wage, and hours.

| Clock | Wage | Hours |
|---|---|---|
| 98401 | 10.60 | 51.0 |
| 526488 | 9.75 | 42.5 |
| 765349 | 10.50 | 37.0 |
| 34645 | 12.25 | 45.0 |
| 127615 | 8.35 | 0.0 |

The program should query the user for how many sets of test data there are, and loop that many times. See the "prompt" code example in the lecture notes on the For Loop for details on how to do this. Do not expect a nice output that shows all the employee data in a one table. It is OK to have your output show the prompts for data on the first employee, print information about that employee, prompt for information about the next employee, print info on that next employee, and so one until each employee data set is processed. You will run this program ONE time and as stated before ... so one will need to initially prompt the user for the number of data sets to process ... which you can then incorporate into your loop test.

Below are some additional items to keep in mind when doing this assignment:

1. Assume that clock numbers are at most 6 digits long and pad with leading zeros when printing if less than 6 digits.
2. Understand that you can't type in the leading zeros for your input, for example, typing in 098401 results in that number being known to the compiler as an Octal number (recall it starts with a zero), and worst, it is invalid because it contains the digits 8 and 9 (only 0-7 are valid). Type in the number 98401 and print with leading zeros to get around this issue. See the "Sample Output" section below to better illustrate what I am requesting.
3. Zero suppression of float fields is important, use the formatting syntax shown in your lecture notes. For example, show just two digits past the decimal point for money fields such hourly wage and gross pay, and only a single digit pass the decimal for the hours field. Without any formatting, the output will show the default of six decimal places (such as: 51.000000 or 551.274321) which does not make for readable output.
4. <u>Do not</u> use any material from any chapters beyond this week's lecture notes. I

realize you could use advanced topics like arrays, pointers, linked lists, strings, structures, and functions ... but part of the challenge is using only what we covered so far.   Rest assured that future homework will incorporate all these advanced topics in the weeks that follow.

5. You may use file pointers if you wish, but it is purely optional.  Doing so will allow you to print the output in the advanced manner shown in the last section at the end of this page. See the coding template provided in the lecture note that follows to get started with this option.

## Sample Output - Assignment 1

### Option 1 - Typical Run Session (not using IDEOne)

Here is sample output that would display from a standard compiler you would have installed on your computer.   The items in blue indicate the values you would have to type onto the screen when prompted by your program, while the output generated by your program is shown in purple.

```
***Pay Calculator***

Enter number of employees to process: 5

Enter Employee's Clock #: 98401
Enter hourly wage: 10.60
Enter number of hours worked: 51.0

-----------------------------------------
Clock# Wage Hours Gross
-----------------------------------------
098401 10.60 51.0 540.60

Enter Employee's Clock #: 526488
Enter hourly wage: 9.75
Enter number of hours worked: 42.5

-----------------------------------------
Clock# Wage Hours Gross
-----------------------------------------
526488 9.75 42.5 414.38

Enter Employee's Clock #: 765349
Enter hourly wage: 10.50
Enter number of hours worked: 37.0

-----------------------------------------
Clock# Wage Hours Gross
-----------------------------------------
765349 10.50 37.0 388.50
```

```
Enter Employee's Clock #: 34645
Enter hourly wage: 12.25
Enter number of hours worked: 45.0

----------------------------------------
Clock# Wage Hours Gross
----------------------------------------
034645 12.25 45.0 551.25

Enter Employee's Clock #: 127615
Enter hourly wage: 8.35
Enter number of hours worked: 0.0

----------------------------------------
Clock# Wage Hours Gross
----------------------------------------
127615 8.35 0.0 0.00
```

## Option 2 - If using IDEOne

If you use IDEOne, you must enter all input items up front, locating them in its "stdin" area. Below is the set of sample inputs you are welcome to use. This example is telling the program to "process 5 employees", and then provides the Clock Number, Hourly Wage, and Number of Hours worked for each of the 5 employees. Data is normally delimited by either a blank space or new line for the scanf statement in your program to work correctly.

```
5
98401 10.60 51.0
526488 9.75 42.5
765349 10.50 37.0
34645 12.25 45.0
127615 8.35 0.0
```

If you use IDEOne, you <u>will not</u> see input values after each prompt because all data is entered up front. With other compilers (see previous example), you will be entering those values in real time. Here is sample IDEOne output that would display.

```
***Pay Calculator***

Enter number of employees to process:

Enter Employee's Clock #:
Enter hourly wage:
Enter number of hours worked:

----------------------------------------
Clock# Wage Hours Gross
----------------------------------------
```

```
098401 10.60 51.0 540.60

Enter Employee's Clock #:
Enter hourly wage:
Enter number of hours worked:

-----------------------------------------
Clock# Wage Hours Gross
-----------------------------------------
526488 9.75 42.5 414.38

Enter Employee's Clock #:
Enter hourly wage:
Enter number of hours worked:

-----------------------------------------
Clock# Wage Hours Gross
-----------------------------------------
765349 10.50 37.0 388.50

Enter Employee's Clock #:
Enter hourly wage:
Enter number of hours worked:

-----------------------------------------
Clock# Wage Hours Gross
-----------------------------------------
034645 12.25 45.0 551.25

Enter Employee's Clock #:
Enter hourly wage:
Enter number of hours worked:

-----------------------------------------
Clock# Wage Hours Gross
-----------------------------------------
127615 8.35 0.0 0.00
```

## Advanced Output Display (optional with file pointers)

Here is an advanced way to present the information if you decide to use file pointers (optional) and create an output file that only presents the actual data values, instead of also displaying all the input prompts. In other words, you get a "cleaner" output. You can only get this type of output if you use "File Pointers" that I presented in the looping notes. Doing this is purely optional and will not result in any additional points, yet is a good way to challenge yourself.

Please note that you can not use file pointers with IDEOne ... it does not support it because it will not let you write a file in its cloud environment. Understand in this scenario, you still prompt for values, it is just that they will show up on the screen (which in C terms is called: ***stdin***), while the items below will be routed to a designated output file on your computer ... thanks to using your output file pointer.

```
 Clock  Wage Hours Gross  Pay
------ ---- ----- --------
098401 10.60  51.0    540.60
526488  9.75  42.5    414.38
765349 10.50  37.0    388.50
034645 12.25  45.0    551.25
127615  8.35   0.0      0.00
```

# Assignment 1 - Code Templates

Below are two C program templates to help you ***get started*** with your ***first homework*** assignment. They show much of what I am looking for according to the class coding standards I published this week. Both examples are missing a loop to process the five employees referenced in the homework assignment. The second code example that follows adds a ***file pointer*** that can be used to actually create an output file on your computer. Note that file pointers will not work if you are using a web based compiler like IDEOne or codepad (its not going to let you create a file on their server), but will work well if you have installed a compiler natively on your computer (Visual C++, Visual Studio, XCode, CodeBlocks, NetBeans, others).

## Non File Pointer Example

Here is a good program template that compiles and works out of the box to get you started with homework assignment 1.   The code is ***missing*** two things:

1) A ***prompt*** (i.e., a printf and scanf statement) to read in the number of employees to process.  An example of this is shown in the **For Loop** lecture note.
2) A ***loop*** to process that many employees (that you prompted for).   You do not have to use a **For Loop**, you can alternatively use a **While Loop** or a **Do Loop**.

Feel free to use this template or the file pointer code that follows to help you get started.  Note I did mix and match C and C++ comment styles.  Just be as consistent as you can with them.   In my case, I used the C++ style comments for the file header, and used C style comments within the code.    You can also see it work with input for one employee as well as use this link as a starting point at:  http://ideone.com/ePMG7t

```
//**********************************************************
//
// Homework: 1 (Chapter 4/5)
//
// Name: Timothy Niesen
//
// Class: C Programming, Fall 1996
//
// Date: 9/19/96
//
// Description: Program which determines gross pay and outputs
// be sent to a designated file.   This version does not use file pointers.
//
//
//**********************************************************

#include <stdio.h>
```

```
int main ( )
{

    int clock_num; /* employee clock number */
    float gross;      /* gross pay for week (wage * hours) */
    float hours;      /* number of hours worked per week */
    float wage;       /* hourly wage */

    /* ADD YOUR PROMPT and LOOP CODE HERE */

    /* Prompt for input values from the screen */
    printf ("This is a program to calculate gross pay.\n");
    printf ("You will be prompted for employee data.\n\n");
    printf ("Enter clock number for employee: ");
    scanf ("%d", &clock_num);
    printf ("Enter weekly wage for employee: ");
    scanf ("%f", &wage);
    printf ("Enter the number of hours the employee worked: ");
    scanf ("%f", &hours);

    /* calculate gross pay */
    gross = wage * hours;

    /* print out employee information */
    printf ("\n\t\tTim Niesen, C Programming, First Homework Assignment\n\n\n");
    printf ("\t----------------------------------------------------------\n");
    printf ("\tClock # Wage Hours Gross\n");
    printf ("\t----------------------------------------------------------\n");

    printf ("\t%06i %5.2f %5.1f %7.2f\n",clock_num, wage, hours, gross);

    /* END YOUR LOOP HERE */

    return (0); /* success */

} /* main */
```

## File pointers example

The example below is also a great start for your first homework assignment that is posted at the end of the lecture notes this week. It adds something we haven't talked about, **file pointers**. In the example below, a file pointer called **outputfileptr** is created that will point to a new file called **home1.txt.** If a file with that name already existed, it would simply be overwritten. You don't need to understand everything at this point (The "if" statement will be explained next week, and the rest in Chapters 11 and 17), but you are welcome to try it out and use it in your homework if you wish.

The key point here is that output can be sent to a **file** as well as the **terminal screen**. Instead of using printf, you use **fprintf**, which adds the file pointer (*outputfileptr*) as one of its arguments. Feel free to use file pointers on your homework assignments. There is no extra credit for using them, but I just wanted to make you aware of how to use them.

At a minimum in the code below, notice how I've indented the code, added a header at the top in comments, declared descriptive variable names in alphabetical order, used sufficient comments, and added blank lines between logical areas.   These simple things can really make your program both highly **readable** and **maintainable**.

**NOTE:** You need to use file pointers with a real compiler installed on your computer, **you can't do them with a web type compiler like IDEOne,** since it will have no place to write the file on its server system.  You can see it run, but unable to process the file pointer, at: http://ideone.com/RSE4Fh.   If you run this code on a local compiler, than it will create the file home1.txt on your hard drive.

```
//********************************************************
//
//   Homework: 1 (Chapter 4/5)
//
//   Name: Timothy Niesen
//
//   Class: C Programming, Fall 1996
//
//   Date: 9/19/96
//
//   Description: Program which determines gross pay and outputs
//   be sent to a designated file.
//
//
//********************************************************

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

int main ( )
{
    int clock_num;          /* employee clock number */
    float gross;            /* gross pay for week (wage * hours) */
    float hours;            /* number of hours worked per week */
    FILE *outputfileptr;    /* pointer to the output file */
    float wage;             /* hourly wage */

    /* open a file called home4.txt */
    if ((outputfileptr = fopen("home1.txt", "w")) == (FILE *) NULL)
    {
       fprintf(stderr, "Error, Unable to open file\n");   /* stderr will print to the screen
*/
       exit(1);
    }
    /* ADD YOUR PROMPT and LOOP CODE HERE */

    /* Prompt for input values from the screen */
    printf ("This is a program to calculate gross pay.\n");
    printf ("You will be prompted for employee data.\n\n");
    printf ("Enter clock number for employee: ");
    scanf ("%d", &clock_num);
    printf ("Enter weekly wage for employee: ");
    scanf ("%f", &wage);
    printf ("Enter the number of hours the employee worked: ");
    scanf ("%f", &hours);

    /* calculate gross pay */
    gross = wage * hours;

    /* print out employee information to a file */
    fprintf (outputfileptr, "\n\t\tTim Niesen, C Programming, First Homework
Assignment\n\n\n");
    fprintf (outputfileptr, "\t-------------------------------------------------------\n");
    fprintf (outputfileptr, "\tClock # Wage Hours Gross\n");
    fprintf (outputfileptr, "\t-------------------------------------------------------\n");

    fprintf (outputfileptr, "\t%06i %5.2f %5.1f %7.2f\n",clock_num, wage, hours,
gross);

    /* END YOUR LOOP HERE */

    return (0); /* success */

} /* main */
```

## Microsoft Visual Studio™ Notes

If you are using any of the Microsoft compilers, like Visual Studio or Visual C++, you will likely get warnings when using **scanf** and **fopen** functions, which will direct you to use their "safer" functions instead, ***scanf_s*** and ***fopen_s***.  You can still run your program with these warnings, but if you do not want to see them, just add the following symbolic constant to the top of the file after your include statements.  It has no value, but makes a "symbol" known to the compiler to suppress warning messages in this area.

```
#define _CRT_SECURE_NO_WARNINGS  /* Visual Studio, turn off scanf,
fopen warnings */
```

Alternatively, feel free to use the ***scanf_s*** and ***fopen_s*** functions preferred by Microsoft (it's likely they are not available in non-Microsoft compilers).  Details were discussed in first week's lecture notes on how to use scanf_s, but just replace ***scanf*** with ***scanf_s***, for example:

```
scanf_s ("%f", &wage);
```

The "safer" part of scanf_s comes into play when we start working with character strings (stay tuned). If you want to use file pointers with ***fopen_s***, below is an example of how you could change my original file pointer template code:

```
/* open a file called home1.txt using fopen_s */
if ( (fopen_s ( &outputfileptr, "home1.txt", "w") ) != (FILE *) NULL)   /* Open
a file to write to */
{
   fprintf (stderr, "Error, Unable to open file\n");   /* stderr will print to the
screen */
   exit (1);   /* Exit the program with error flag set  */
}
```

# Assignment 1 Notes

Start your first homework assignment, it will be due by next Monday at midnight. All I need is the source code which you can upload a C file or paste the contents into the Assignment 1 drop submission area (in the Assignment 1 link at the end of the looping lecture notes).     If you use IDEOne, then just paste in the URL to your code.

DO NOT SEND YOUR HOMEWORK TO THE DISCUSSION BOARD :)

... however, feel free to post partial code if you have a question on some of your code ... just not ALL your code

One other important thing, please review and utilize the **homework standards** that are available near the bottom of the class page (scroll down a bit). Some of my grading is based on your adherence to this standard and it is important for readability and maintainability ... you will thank me at the end when you see how good your programs can look and feel :)

I will then grade it where you can view it anytime along with my comments. Most grades range between 80-100 for the most part if you follow the directions. You have a great chance of doing well in this course if you keep up with the homework and not fall behind. I will mark down for lateness. If you have any problems, do not hesitate to email me, that's what they pay me the big bucks for! **You need only send me the source code.**

Note that your output won't look that great, unless you use **file pointers** (which is **optional** ... and file pointers will not work with IDEOne).    Its OK to have your output show the prompt information for an employee, and then print the information, then show the prompt information for the next employee and print their information, and so on until the last employee is processed.

Here is a shell of the homework problem if you were starting from scratch (but feel free to use the templates above to get started).

```
#include <stdio.h>
int main ( )
{
    /* Declare variables */

    /* Prompt for number of employees to process */

    /* Use any type of loop, repeat until all employees processed */

        /* Prompt for input on a single employee from the screen */
        /* Calculate gross pay */
        /* Print out employee information to the screen */

    /* end the loop */

    return (0);

} /* end main */
```

## Video - Submitting Work for Grading

This is a short video for those new to online classes. It just demonstrates how a student submits a file or set of files for grading, for either a homework assignment or an exam. It also shows what I see as a teacher, and how I review and record a grade.