# Welcome to Week 3!

**This week you should ...**

## Required Activities

- Go through the various **videos, movies, and lecture notes** in the **Week 3 folder**.
- Read **Chapter 5** in the latest edition of the textbook (chapter 6 in earlier editions).
- Begin **Quiz 3**. It is due Sunday at midnight.
- Begin **Assignment 2**. It is due Sunday at midnight.

## Recommended (optional) activities

- Attend **chat** this Thursday night, from 8:00 pm - 9:00 pm Eastern Time. Although chat participation is optional, it is highly recommended.
- Post any questions you might have on this week's topic in the **Week 3 Discussion Forum** located in the course **Discussion Board**. Please ask as many questions as needed, and don't hesitate to answer one-another's questions.
- Try out the various **code examples** in the lecture notes. Feel free to modify them and conduct your own **"What ifs"**.

*"Kurt, I can guarantee you this ... If you cross the Neutral Zone, you WILL be violating the treaty between us."*

--- Annoying Klingon "Korn"
Captain, IKS Bird of Prey Akua

# Conditional Processing

This week we'll learn about conditional processing. The essence of good computer programming is to factor into your design the ability to handle all possible **conditions** that could take place.   You need to continuously ask yourself:  "What could happen next?".

For example, if a user clicks a specific button on a web form, a corresponding set of code will be executed. Sometimes there are alternatives, such as if I am 50 years or older, I can join the American Association of Retired Persons (AARP), otherwise I can not join. There are only two possible conditions here, you either are eligible to join AARP or not ... there is no middle ground.

Of course, in many cases, there are **different alternatives** based on **multiple conditions**.  Programming Languages like C allow you to easily design and implement your code so that all **possible conditions** can be taken into account.

# Star Trekking - Negotiating Tough Decisions

To introduce you to the concept of conditional processing, let's look in on a negotiation that is going on in a galaxy far, far away. In this movie, Captain Kurt of the U.S.S. Enterprise is deciding on whether or not to enter the dreaded "Neutral Zone", thus violating a treaty between Star Fleet and the Klingon Empire.  Violation of this treaty could have various consequences.

Access the movie by clicking the image posted below or the link on the next lecture note page to watch how various conditions were negotiated to come up with an acceptable solution for all sides. Note that if you were developing a program to simulate what could happen, you would have to factor in all possible scenarios.

# Answers to Last Week's Exercises

Below are my answers to the selected exercises I posted at the end of last week's lecture notes. You should be able to easily copy and paste them into your favorite compiler/development environment. Once you do that, feel free to try some variations of the programs by tinkering and doing "what ifs" with the code.

## 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.
*/
#include <stdio.h>
main ()
{

  int num;           /* a series of numbers */
  int num_squared;   /* each number squared */

  printf ("TABLE OF SQUARES FOR 1 TO 10\n\n");
  printf (" num      num squared\n");
  printf ("-----     -----------\n");

  /* loop from 1 to 10 */
  for (num = 1; num <= 10; ++num )
  {
      printf ("  %d         %d\n", num, num*num);
  }

  return (0);
}
```

## 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)
*/

/* with a for loop */
#include <stdio.h>
```

```
main ()
{

   int n;         /* a number */
   int tri_num;   /* the triangular value of the number */

   printf ("TABLE OF TRIANGULAR NUMBERS by 5\n\n");
   printf (" n     Triangular Number\n");
   printf ("---    -----------------\n");

   tri_num = 0;

   for (n = 5; n <= 50; n = n + 5)  /* or n+= 5 */
   {
       tri_num = (n * (n + 1))/2;
       printf (" %d        %d\n", n, tri_num);
   }

   return (0);
}


/* with a do loop */

#include <stdio.h>
main ()
{

   int n;         /* a number */
   int tri_num;   /* the triangular value of the number */

   printf ("TABLE OF TRIANGULAR NUMBERS by 5\n\n");
   printf (" n     Triangular Number\n");
   printf ("---    -----------------\n");

   tri_num = 0;

   n = 5;
   do
   {
       tri_num = (n * (n + 1))/2;
       printf (" %d        %d\n", n, tri_num);
       n = n + 5;  /* better:  n += 5; */
   }
   while (n <= 50);

   return (0);
}

/* with a while loop */

#include <stdio.h>
main ()
{
   int n;          /* a number */
   int tri_num;    /* the triangular value of the number */

   printf ("TABLE OF TRIANGULAR NUMBERS by 5\n\\n");
   printf (" n     Triangular Number\n");
   printf ("---    -----------------\n");

   tri_num = 0;

   n = 5;
   while (n <= 50)
   {
       tri_num = (n * (n + 1))/2;
       printf (" %d        %d\n", n, tri_num);
       n = n + 5; /* or n+= 5 */
   }

   return (0);
}
```

## Sample Exercise 3

```
/* Sample Exercise 3

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

/* with a for loop */

#include <stdio.h>
main ()
{

  int num, factorial;

  printf ("TABLE OF FACTORIALS FOR 1 TO 10\n\\n");
  printf (" num      num factorial\n");
  printf ("-----     -------------\n");

  factorial = 1;
  for (num = 1; num <= 10; ++num )
  {
      factorial *= num;
      printf ("  %d         %d\n", num, factorial);
  }

  return (0);
}
```

```
/* with a do loop */

#include <stdio.h>
main ()
{

  int num, factorial;

  printf ("TABLE OF FACTORIALS FOR 1 TO 10\n\n");
  printf (" num      num factorial\n");
  printf ("-----     -------------\n");

  num = 1;
  factorial = 1;
  do
  {
      factorial *= num;  /* or factorial = factorial * num */
      printf ("  %d         %d\n", num, factorial);
      num++;
  } while (num <= 10);

  return (0);
}

/* with a while loop */

#include <stdio.h>
main ()
{

  int num, factorial;

  printf ("TABLE OF FACTORIALS FOR 1 TO 10\n\n");
  printf (" num      num factorial\n");
  printf ("-----     -------------\n");

  num = 1;
  factorial = 1;
  while (num <= 10)
  {
      factorial *= num;
      printf ("  %d         %d\n", num, factorial);
      num++;
  }

  return (0);
}
```

## Sample Exercise 4

```c
/* Exercise 11 from the book

   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.

*/
#include <stdio.h>
main ()
{

    int right_digit, number, sum_of_digits = 0;

    printf ("Enter your number: ");
    scanf ("%d", &number);

    while (number != 0)
    {
        right_digit = number % 10;
        printf ("right digit = %d", right_digit);
        sum_of_digits += right_digit;
        number = number / 10;
        printf (", number = %d\n", number);
    }

    printf ("\n");
    printf ("Sum_of_digits = %d\n", sum_of_digits);

    return (0);
}
```

# Video - C Conditionals

This video shows you how to work with the basic conditional statements in C. It covers how to properly use boolean logic, if statements, if else statement, and else if statements. Complete animation is provided to illustrate exactly how this is being done within C.

# Conditional Processing

There are **three fundamental properties** of computer programming:

1) Sequential execution of statements
2) Looping - Repetitively execute a sequence of statements
3) **Decision Making** - Conditionally execute a sequence of statements

I would argue that if you have a programming language that can process all three of these properties, that you can design and implement a program to do anything that needs to be done. We've already shown you the first two properties, lets look at the last one to conditionally execute a sequence of statements.

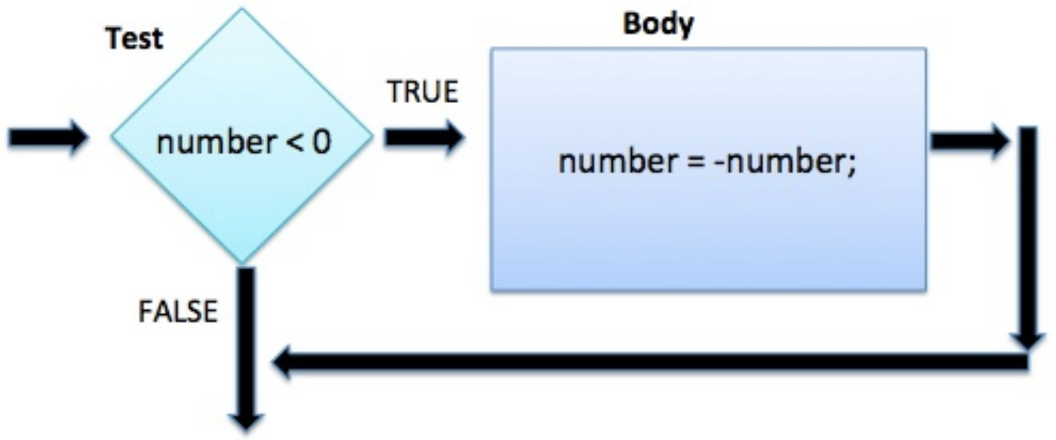
When you put together a sequence of statements to be executed, you need the ability to selectively execute statements based on one or more conditions. For example, if I was determining grades for an exam, I would say that a student passed if they got a score of 60 or higher. However, if you really think about it, a student either passes OR fails an exam, there really is no middle ground. The first example I posed in considering only whether a student passed is an example of a simple **"if"** statement, while the latter is an example of an **"if else"** statement. Let's look at exactly how this is accomplished in the C Programming Language, and then continue with other conditional processing type statements in C. The important thing to understand here is that the basic logic of these statements in the C Language is essentially the same logic (and in most cases, code syntax) that is employed by most modern languages in use today.

# The if Statement

The if statement has the following syntax

```
if ( expression )
    BODY;

if ( number < 0 )
    number = - number;  /* make it positive */
```

The if statement above could be envisioned as:



You have to use parentheses around your expression, they are mandatory, don't do this:

```
    if  number < 0     /* Parentheses are mandatory */
```

The **expression** may use a relational operator (<,>,<=,>=,==,!=) or it may be a mathematical expression such as a + b. If the expression equates to non-zero, it is considered TRUE; otherwise FALSE. The **body** will only be executed if the expression is TRUE.

The program **body** can be one statement, or it can be several statements enclosed within braces. If one statement is in the body, then curly braces are not needed. If multiple statements are in the body, then curly braces are needed. The syntax rules are similar to that of the loops we studied last week.

```
    if ( expression )
    {
        statement1;
        statement2;

        ... statement N;
    }
```

Just like with loops, its a good idea to always enclose the body within curly braces.

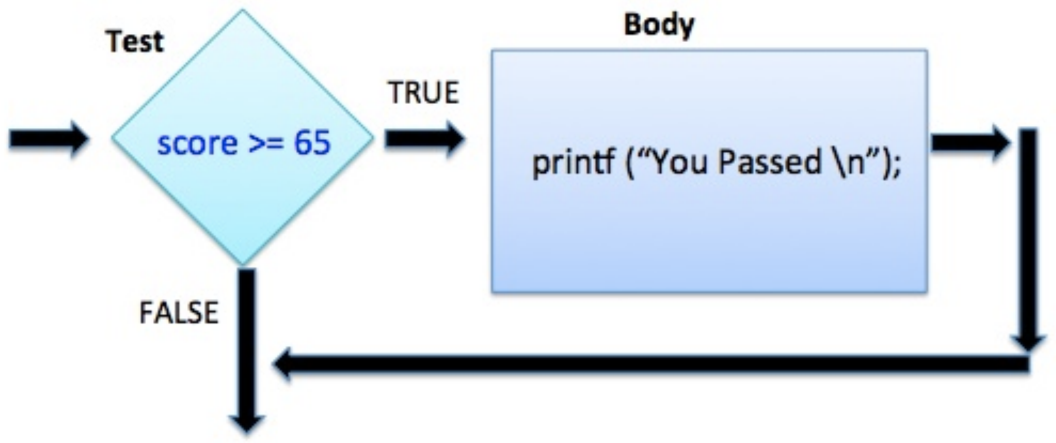
Don't make the rookie mistake of putting a semicolon after the if statement, it will likely result in ignoring all statements in the if body if the expression is TRUE ... just put it instead after each statement in the body ... don't do this:

```
    if ( expression ) ;  /* don't put a semicolon here */
    {
        statement1;
        statement2;

        ... statement N;
```

```
}
```

Also remember that the Parentheses are mandatory for any expression using an if statement. In our exam code example, the if statement would only process a student who passed, otherwise, nothing would happen.

```
if ( score >= 65 )
    printf (" Great Job ... you passed \n");
```



## The if else Statement

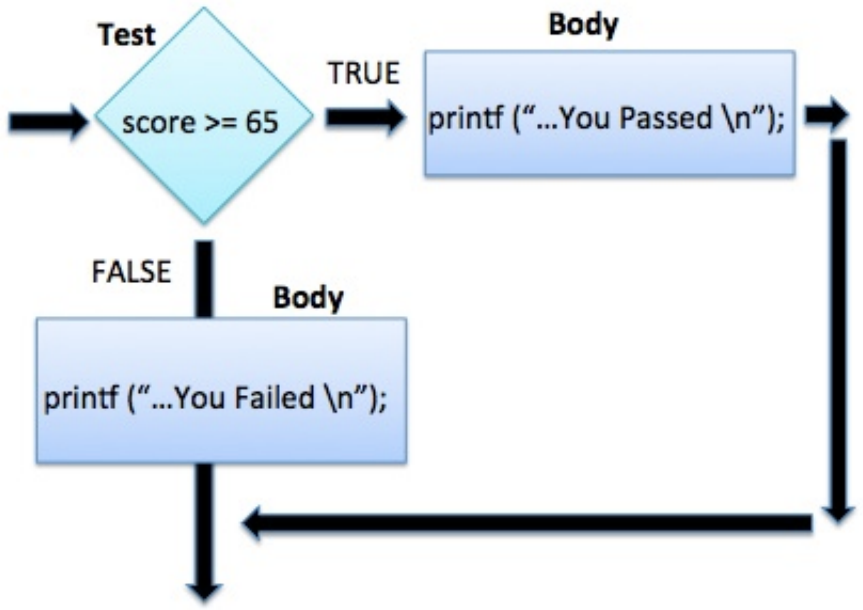
The if else statement has the following syntax

```
if ( expression )
    BODY;
else
    BODY;
```

**Example:**

```
if ( score >= 65 )
    printf ("... You Passed \n");
else
    printf ("... You Failed \n");
```

Here is an illustration on how an if else statement would work:



The **expression** is evaluated. If it is TRUE or non-zero, the **BODY** is executed in the IF, otherwise the BODY in the else statement is executed. This is an ***either or*** situation.

## HINT

Even if there is only one statement in the body, it is a good idea to always enclose it within curly braces in case you add to it later and forget to put them in. Don't forget to indent the body.

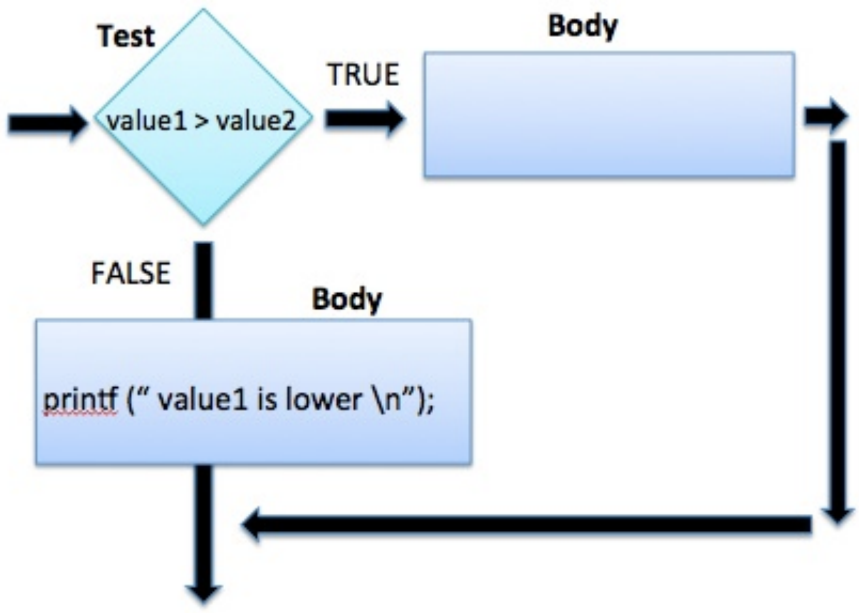
```
if ( score >= 65 )
{
        printf (" You Passed \n");
}
else
{
        printf (" You Failed \n");
}
```

The if else statement has the following syntax

```
if (expression)
    ;

else
    BODY;
```

An example would be:

```
if ( value1 > value2 )
    ; /* do nothing */

else
    printf (" value1 is lower \n");
```



To do nothing when the expression is FALSE, use **if** without **else**. To do nothing in the TRUE case a semicolon with no statement may be used. This allows you to reverse the test from negative to positive. Or use '!' to negate the operation.

To conclude, if you wanted to process our original problem with the exam score, if would be something like this:

```
if  ( score >= 65 )
    printf ("You passed! \n");
else
    printf ("You failed! \n");
```

As stated before, its really best to put the if or else body of statements between **curly braces**.   That statements below do exactly the same thing as the statements above. The below statements are more readable and if you decide to add statements later to the if or else bodies, so you will not potentially introduce an error because you forgot to enclose two or more body statements within curly braces.

```
if  ( score >= 65 )
{
      printf ("You passed! \n");
}
else
{
      printf  ("You failed! \n");
}
```

# Boolean Logic

The principle of **Boolean Logic** lets you organize concepts together in **sets**. You probably do this all the time with simple text base searches into the various popular search engines on the Internet, many of which have advanced searches that let you use the AND and OR logical operators. Let's say you wanted to search for Internet radio stations that played Jazz and Blues. If you wanted to find stations that played BOTH Jazz and Blues, you would use an AND statement:

**Jazz AND Blues**.

What if you wanted a station that played either Jazz or Blues? How about searching:

**Jazz OR Blues**

An OR will **broaden a set**, in our example above, it would give radio stations that play Jazz only, play Blues only, and play both Jazz and Blues. Contrast that with an AND, which will **narrow the set** returned, in our example it would only return radio stations that play BOTH Jazz and Blues.

Note while there are other Boolean logic type operators (for example, **NOT** is an important one), let's just confine our discussion to OR and AND.

# Logical OR

The logical OR operator is used to evaluate relational expressions that will evaluate to TRUE or FALSE. It is expressed in C as two vertical bar characters || (not the number 1 or the letter l). Custom puts the two symbols together:

**||** ... not ... **| |**

A table is shown below that shows the possible combination values of TRUE and FALSE values. The easiest way to think about an OR operator in a expression is if just one part is TRUE, the result is TRUE. The only way for it to be FALSE is if all conditions are FALSE.

```
if (index < 0 || index > 99)
```

| OR (\|\|) | RESULT |
|---|---|
| True \|\| True | True |
| True \|\| False | True |
| False \|\| True | True |
| False \|\| False | False |

With the **or** operator, the **if** test will fail only if both parts fail. One common mistake new C programmers make is trying to take a few shortcuts that will not always work. Don't code something like this below.

```
if ( value < 0 || > 99 )
```

Instead, the variable **value** must be used for each check in the if statement:

```
if ( value < 0 || value > 99 )
```

---

# Logical AND

The logical AND operator is used to evaluate relational expressions that will evaluate to TRUE or FALSE. It is expressed in C as two & characters: **&&**

Custom puts the two symbols together:

**&&** not **& &**

A table is shown below that shows the possible combination values of TRUE and FALSE values: The easiest way to think about an AND operator in an expression is the only way for a result to be TRUE is if all conditions in an expression are TRUE.

```
if (index < 0 && index > 99)
```

| AND (&&) | RESULT |
|---|---|
| True && True | True |
| True && False | False |
| False && True | False |
| False && False | False |

- if (value < 0 && > 99) is not valid ... just like with the OR example
- With the **and** operator, the **if** test will succeed only if both parts succeed (i.e., are TRUE). The **&&** has higher precedence than **||**, lower precedence than arithmetic and relational operators.

  ```
  if ( index > 5 || b == 5 && j < 34 )
  ```

  will be interpreted by the compiler as:

  ```
  if ( index> 5 || ((b == 5) && (j < 34)) )
  ```

- To avoid uncertainly, use parentheses for complex statements
- To aid readability, leave a space on either side of the operator

NOT: if (index < 0&&index> 99)

---

# Logical AND and OR

Be careful about using || and &&. For example, what if you had to determine if a number was between 0 and 100?

RIGHT

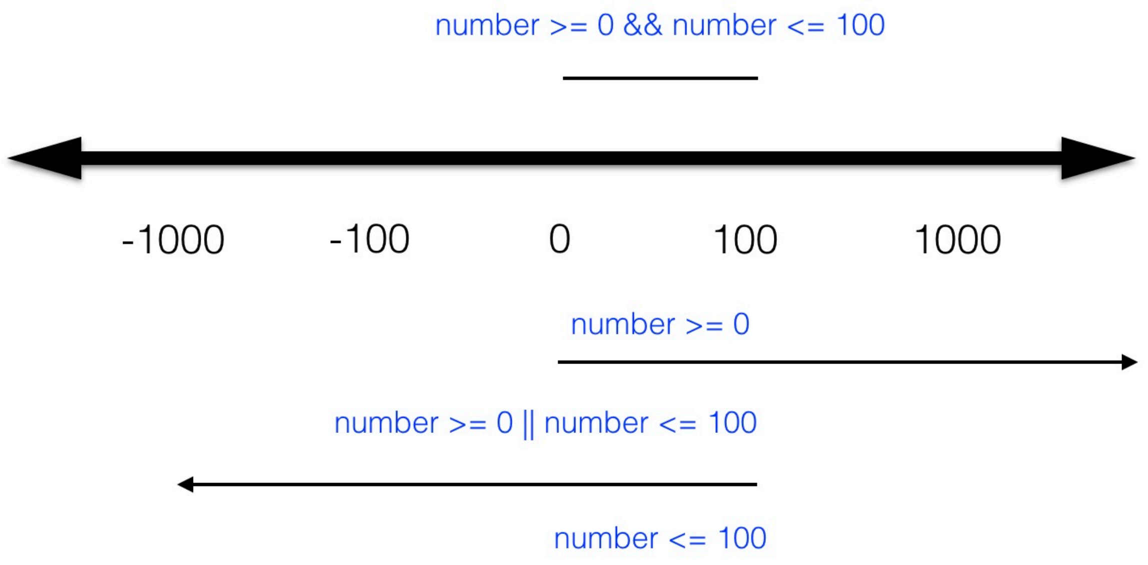
```
if ( number >= 0 && number <= 100 )
   printf ("Between 0 and 100");
else
   printf ("Not between 0 and 100");
```

WRONG

```
if ( number >= 0 || number <= 100 )
   printf ("Between 0 and 100");
else
   printf ("Not between 0 and 100");
```

- THIS WILL BE TRUE FOR **ALL NUMBERS** !

To make this a bit clearer ... consider the number line below that illustrates what exactly is happening. With the OR statement, essentially all numbers would be true. With the AND statement, you can restrict the values between 0 and 100.



Finally, you can have a rather large set of complex statements with these operators. In this case, using parentheses can really help make things more understandable for the coder and the person maintaining the code.

```
if   ( ( a == 6 ) || ( b == 5 && c == 9 ) ||
       ( k >= 5 || ( j <0 && b < 0) ) &&
       ( z == 9) )
```

## Nested if

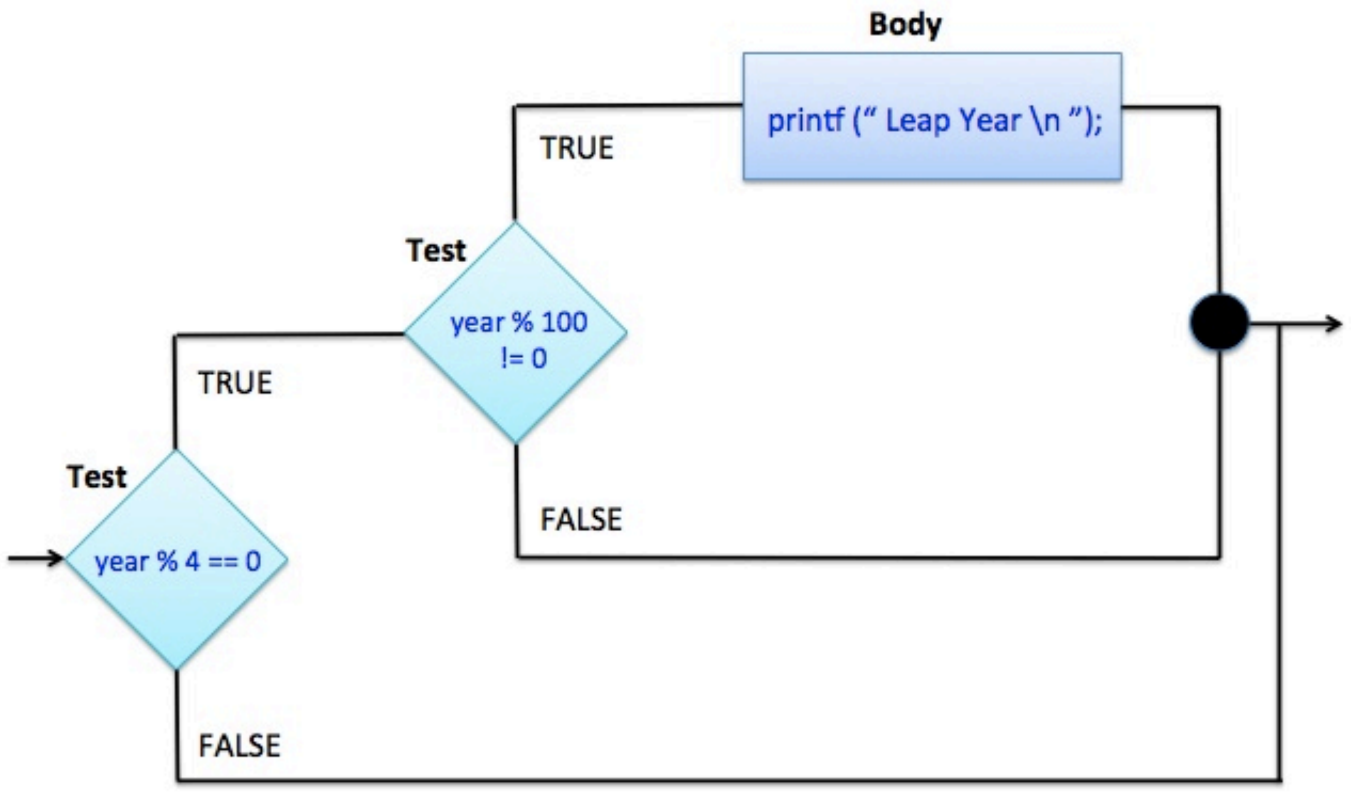
A if statement may be shown inside another if statement. In this scenario, the **inner if** statement is said to be **nested** inside the **outer if** statement. The **nested if** is ***equivalent*** to the logical operator **AND**. You will not reach the second test (the nested if) if the first if statement fails. Note how the below code has a **nested if** within the **first if** statement.

```
if ( year % 4 == 0 )     /* this is the first if statement */
   if ( year % 100 != 0 )   /* this is my nested if */
      printf ("Leap Year \n");
```

***COMPARE TO:***

```
if ( year % 4 == 0 && year % 100 != 0 )
   printf ("Leap Year\n");
```

A **flowchart** illustrating the nested if statement below would look like this:



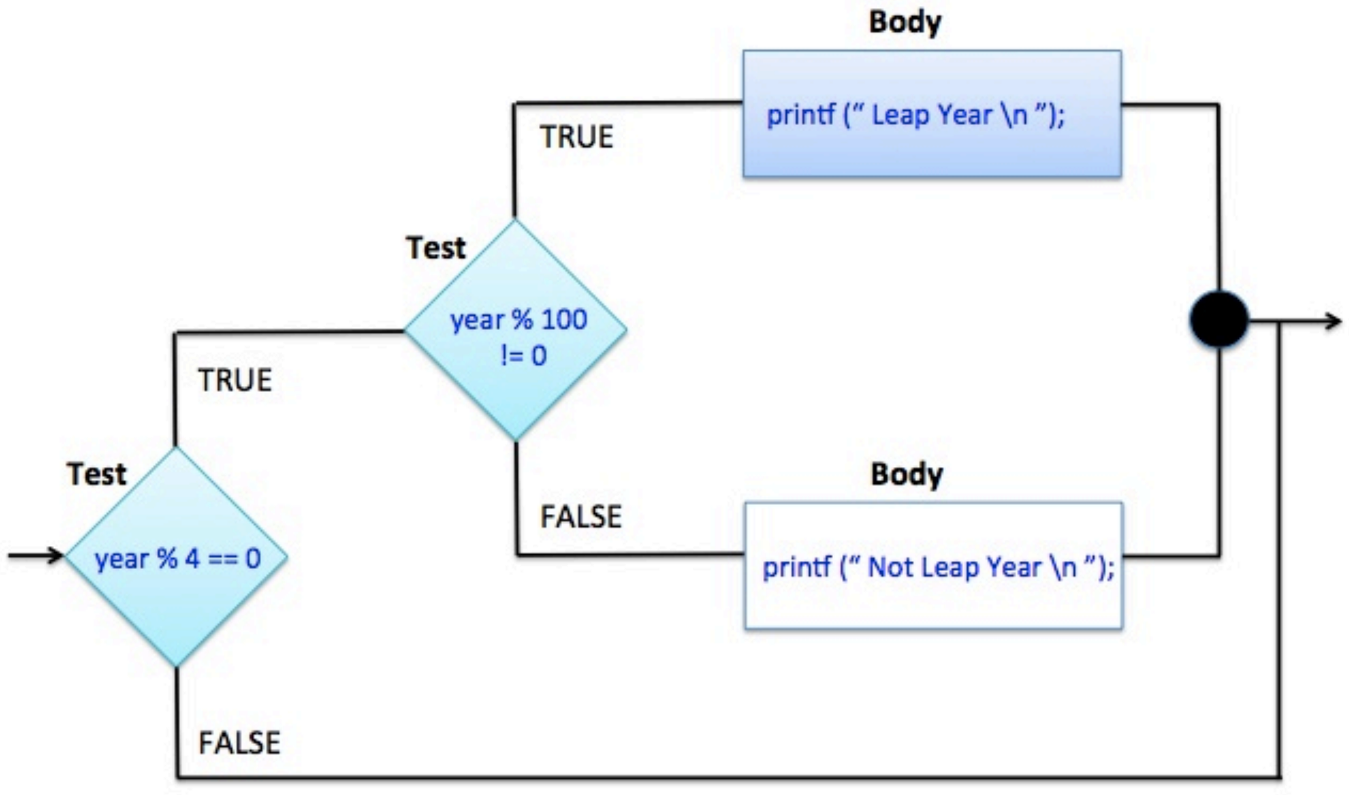
Try not to have too many nested if statements. It can be difficult to follow. I find the use of an AND operator to be easier to follow and less prone to mistakes.

## Nested if else

We kind of alluded to this above, but It is also possible in C to **nest if-else statements**, which means you can "nest" an **if else** statement inside an **if** statement. The nested if else statement is shown below in a series of statements that is designed to determine if a year is a leap year.

```
if ( year % 4 == 0 )
   if ( year % 100 != 0 )
      printf ("Leap Year \n");
   else
      printf ("Not Leap Year \n");
```

A **flowchart** that illustrates this is presented below:



An else always goes with the most recent un-elsed if, regardless of indentation.

Your **indentation** should match this relationship

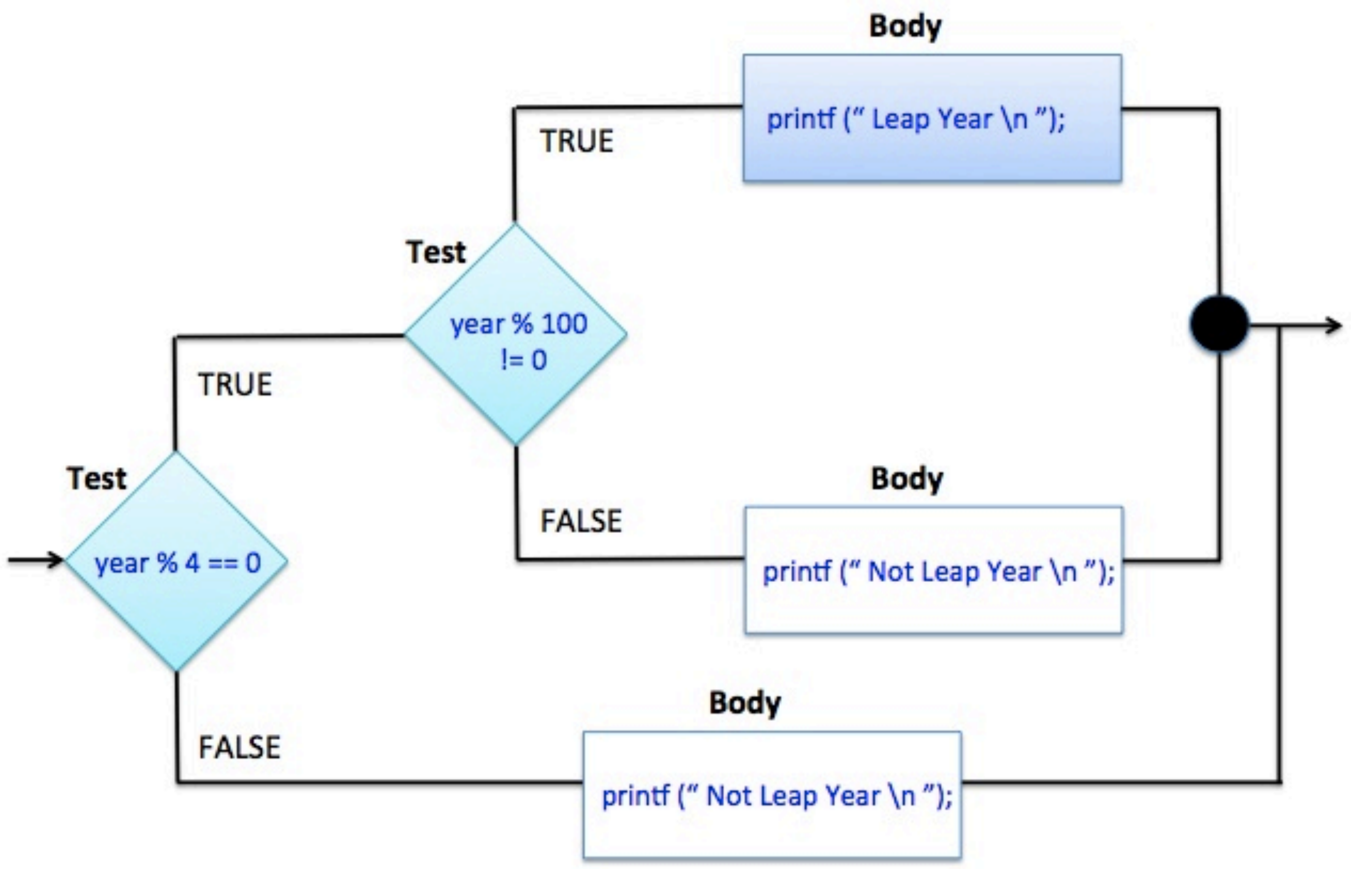
What's wrong with this picture? Nothing is printed if year % 4 doesn't equal 0.

You could also write the **nested if** with a corresponding **else** statement:

```
if ( year % 4 == 0 )
   if ( year % 100 != 0 )
      printf ("Leap Year \n");
   else
      printf ("Not Leap Year\n");

else /* belongs to first if at top */
```

```
        printf ("Not Leap Year \n");    /* duplicate code, not good */
```

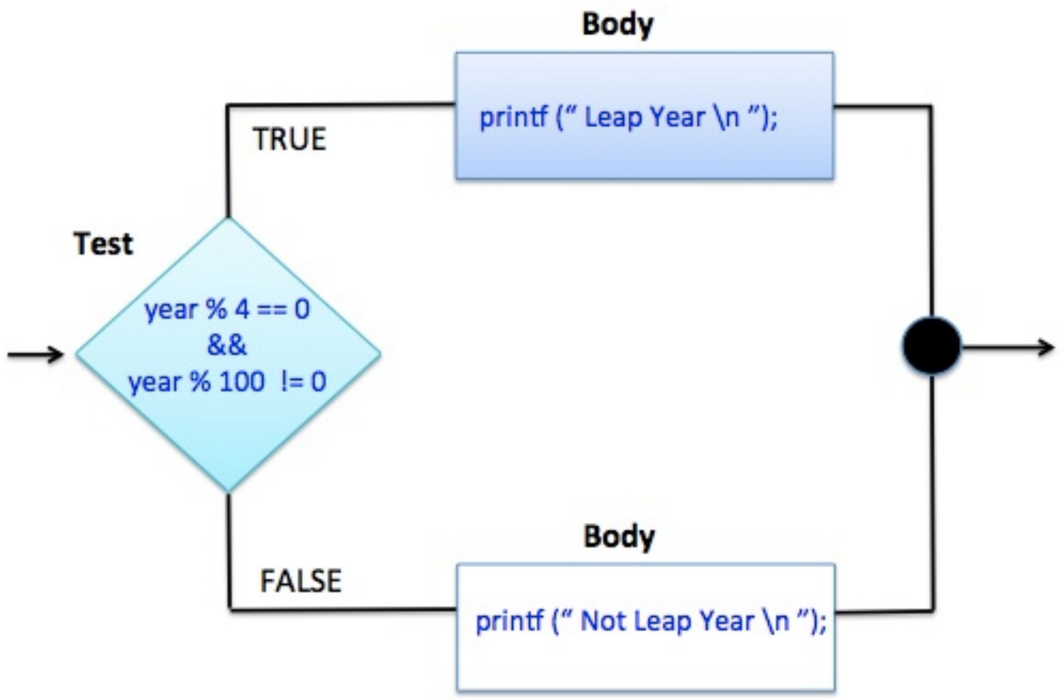
The **duplicate code** is easily identified within a **flowchart** when examining how the ***nested if else*** code above executes:


Body
printf (" Leap Year \n ");
TRUE
Test
year % 100 != 0
TRUE
Test
year % 4 == 0
FALSE
Body
printf (" Not Leap Year \n ");
Body
FALSE
printf (" Not Leap Year \n ");

Compare the **nested if else** example to code below that uses simple **boolean logic** ... both accomplish the same thing in the end. However, I generally prefer using boolean logic as it will generate less paths to follow, and thus be easier to test and generate less machine language code to be executed. Its simple thinking like this that separates a computer programmer from what I do for a living, a software engineer.

```
if ( year % 4 == 0 && year % 100 != 0 )
   printf ("Leap Year \n");
else
   printf ("Not Leap Year \n");
```

Boolean AND statements are not always the answer, but in this case, using one greatly simplified the code. Note how the **flowchart** is now very easy to follow.



**NOTE:** The if to else relationship can be changed by **curly braces**. In the previous nested if example, without the curly braces, the **else** would be connected to the **second if**. By using curly braces, as shown, it can be connected to the **first if** statement.

```
if ( game_is_over == 0 )
{
   if ( player_to_move == you )
      printf ("Your Move \n");
}
else
   printf ("The Game is Over \n");
```

# The else if Statement

In some cases, there are more than just two cases, like a student passed or failed. An **else if** statement allows you to handle multiple cases, and in particular, more than two. Let's look at the **else if** statement in more detail, and then show a detailed example based on a exam score problem.

## SYNTAX

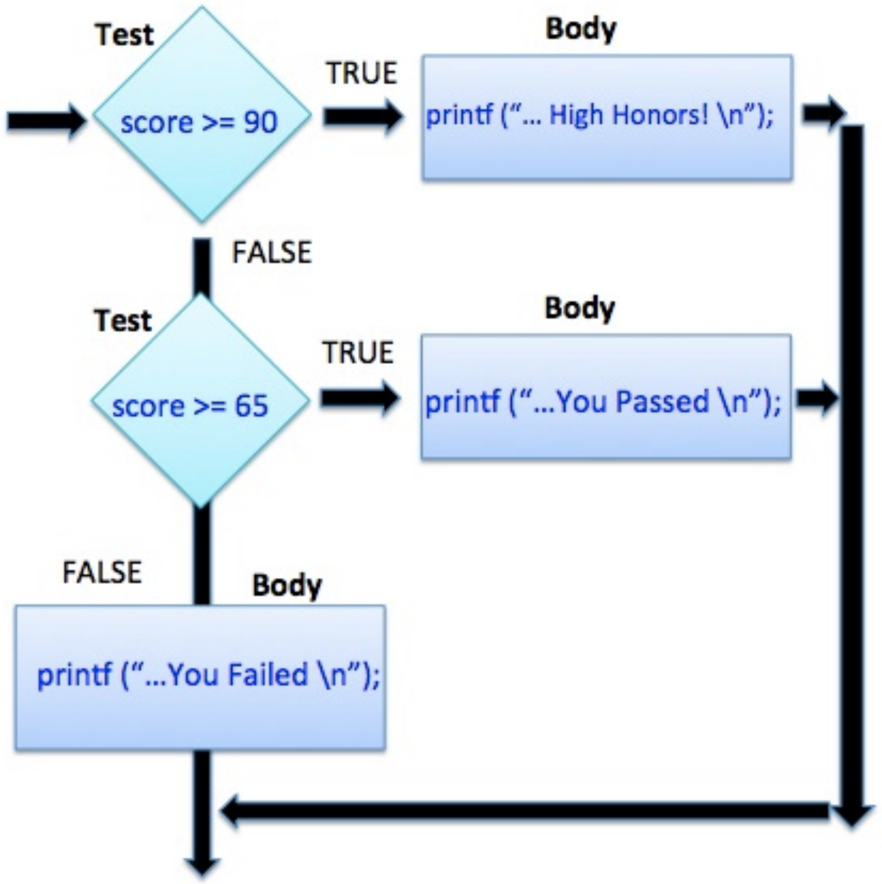
```
if (expression)
     BODY;
 else if (expression)
     BODY;
 else if (expression)
     BODY;
  /* . . .  */
 else
     BODY;
```

## EXAMPLES

```
if (number < 0)
     printf("Negative Number \n");
else if (number > 0)
     printf("Positive Number \n");
else /* optional */
     printf("Zero \n");
```

```
if ( score >= 90 )
     printf (" ... High Honors \n");
else if ( score >= 65 )
     printf (" ... You Passed \n");
else /* optional */
     printf (" ... You Failed  \n");
```

## FLOWCHART


Test
score >= 90
TRUE
Body
printf ("... High Honors! \n");
FALSE
Test
score >= 65
TRUE
Body
printf ("...You Passed \n");
FALSE
Body
printf ("...You Failed \n");

Note: In the flowchart, if its not TRUE, than it must be FALSE

An **if else** provides two possible courses of action.

```
if (expression)
    do action 1;
else
    do action 2;
```

Inserting an else if allows three or more courses of action

There are nested **else if** statements with the ending being an **else** by itself. The **else** statement in an **else if** is optional.

```
if (expression)
    BODY;
else if (expression)
    BODY;

 /* add more else if statements if needed, but else is not mandatory */
 /* ... but if you have an else, there can only be one. */
```

# An "else if" example

Let us now present an example of a program where I could use an **else if** to determine the letter grade based on an exam whose scores range from 0 to 100.   I'll also add some processing to potentially add some money to my savings account :)

The important thing to understand about this program is that an **else if** allows the body of **ONLY ONE** of the **conditions** to be processed.   The initial if statement will check for **invalid states** and the final else statement is a **catch all** for any remaining failure conditions, everything in between will process a passing letter grade.   The trick here so you don't add lots of extra checks between grade ranges is to first eliminate all the invalid scores.   Once you do that, you can then just check for valid scores between 0 and 100.

Another thing to observe in the code is how I **commented** the **ending braces** for each else/else if statement.   Little things like this can really make your code much easier to follow.

```
} /* 80 - 89 */
```

Feel free to try it out in IDEOne at:  http://ideone.com/JbFQXs

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

    char answer[20];   /* user response to yes or no question */
    int score;         /* test score */

    printf ("\nEnter a score: ");
    scanf ("%i", &score);

    /* Process Grade and Options based on Score */
    if (score < 0 || score > 100)
        printf ("\nError, Invalid Grade \n");

    else if (score >= 90)  /* 90-100 */
    {
        /* you need these students, but can't make any money from them :) */
        printf ("\nExcellent Job: Grade is an A \n");
    }  /* 90 - 100 */

    else if (score >= 80)  /* 80-89 */
    {
        printf ("\nGood Job: Grade is a B \n");
        printf ("Would you like an A? (y/n): ");
        scanf ("%s", answer);

        /* Come on, answer yes, professor needs a new electronic toy */
        if (answer[0] == 'y' || answer[0] == 'Y')
        {
           /* All right, a customer */
           printf ("\nMake check payable to Tim Niesen for $200\n");
           printf ("... a small price to pay for your education \n");
        }
     }  /* 80 - 89 */

     else if (score >= 70)  /* 70 - 79 */
     {
        printf ("\nFair Job: Grade is a C \n");
        printf ("Would you like an A? (y/n): ");
        scanf ("%s", answer);   /* The & not needed with Arrays, will discuss
this in the future */

        /* time to make some real money, my favorite students */
        if (answer[0] == 'y' || answer[0] == 'Y')
        {
           printf ("\nMake check payable to Tim Niesen for $600 \n");
           printf ("... an excellent price to pay for your education \n");
        }

        else  /* maybe the student will pay to get a B ? */
        {
           printf ("\nWould you like an B? (y/n): ");
           scanf ("%s", answer);

           if (answer[0] == 'y' || answer[0] == 'Y')
           {
              printf ("\nMake check payable to Tim Niesen for $400 \n");
              printf ("... a great price to pay for your education \n");
           }
        }

     }  /* 70 - 79 */

     else if (score >= 60)  /* 60 - 69 */
     {
        /* just can't make any money here, would not be right */
        printf ("\nPoor Job: Grade is a D \n");
        printf ("... You're beyond a bribe to get a better grade \n");
     }  /* 60 - 69 */

     else  /* < 60 */
     {
        printf ("\nYou Failed: Grade is an F :(\n");
     }
```

```
    return (0); /* success */

} /* end main */
```

Please note that the above example is just for fun ... don't be sending me any checks this semester :)

A **bad design** of the program above would be implementing it instead with a **series of if statements**.   If they were ALL **if statements**, you would have to check **every condition**.   Once the program determines that a grade is an A, it does not need to check all the other conditions.    A properly defined series of else if statements is very **efficient**.    See the code I bolded and highlighted in RED below that shows what you should not do.

```
/* the first if statement is OK */
if (score < 0 || score > 100)
   printf ("Error, Invalid Grade \n");

if (score >= 90)     /* BAD ... this should be an else if */
{
   /* you need these students, but can't make any money from them :) */
   printf ("Excellent Job: Grade is an A \n");
}

if (score >= 80)     /* BAD ... this should be an else if */
{
```

# else if in a Loop

Below is an example of a program that will read in two number and an arithmetic operator and evaluate an expression based on the input. Take a close look at the example below and free to also try it out at: https://ideone.com/A7U9F5

```
/* Program to evaluate simple expressions of the form:   number operator number
*/

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

   float value1;   /* floating point value to read              */
   float value2;   /* another floating point value to read */

   char answer;          /* To determine if more input is available                */
   char myOperator;   /* The operator to run against our two input values */

   /* process expressions until the user types a 'n' character */
   do
   {

      printf ("\nType in your expression.\n");
      scanf ("%f %c %f", &value1, &myOperator, &value2);

      /* process our two values based on the operator specified */
     if ( myOperator == '+' )
        printf ("%.2f\n", value1 + value2);
     else if ( myOperator == '-' )
        printf ("%.2f\n", value1 - value2);
     else if ( myOperator == '*' )
        printf ("%.2f\n", value1 * value2);
     else if ( myOperator == '/' )
        printf ("%.2f\n", value1 / value2);

     printf ("\nWould you like to enter another expression? (y/n) ");
     scanf (" %c", &answer);

   }  while ( answer != 'n' );

   printf ("\nGoodbye");

   return 0;
}
```

## Input

```
5 + 3
y
7 * 4
y
10 / 2
y
6 - 2
n
```

## Output

```
Type in your expression.
8.00

Would you like to enter another expression? (y/n)
Type in your expression.
28.00

Would you like to enter another expression? (y/n)
Type in your expression.
5.00

Would you like to enter another expression? (y/n)
Type in your expression.
4.00

Would you like to enter another expression? (y/n)
Goodbye
```

## A Word about Reading a Single Character in a Loop

One of the problems with just reading a character in a loop by itself, with either a getchar or scanf statement, is the presence of the newline character ( '\n'). The key here is to consume the newline character if present before reading in the next character. This is how I accomplished it in the program above.

```
scanf (" %c", &answer);  /* adding a space before will do the trick in the format */
```

If you it code it this way, it won't work and you will get an infinite loop as the newline character will get in the way each time.

```
scanf ("%c", &answer);  /* newline character won't be consumed */
```

... this way won't work either as the space must be before the %c in the format

```
scanf ("%c ", &answer);  /* newline character won't be consumed if space put after the %c */
```

Some other alternative ways to consume the newline character that will work:

```
scanf ("\n");   /* consume the newline character with another scanf  */
scanf ("%c", &answer);

getchar ();   /* consume the newline character with a getchar library function */
scanf ("%c", &answer);

scanf ("\n%c", &answer);  /* consumes the newline character and read a character */

                          /* all within a single scanf statement                 */
```

# The Switch Statement

When the world "switch" come to mind, people normally think of two things, a *light switch* that you turn on or off, and a switch dealing with *train tracks*.   A **switch** in programming terms is more like a switch on a **train track**.  See this quick YouTube video below that illustrates how a train switch could be activated to allow a train to go from one track to another.

Watch Video

**Planet coaster Train Switch Track**

**User:** n/a - **Added:** 11/22/16

Some switches can even be designed to handle three, four, five or more track directions!

In the programming world, a **switch statement** is a selection statement that lets you transfer control to different statements within the **switch body** depending on the value of the **switch expression**.  It does have its limitations as it can only evaluate a single expression.  The value of the switch expression is compared with the value of the **expression** in each **case label**. If a matching value is found, control is passed to the statement following the case label that contains the matching value.

Some would call the Switch Statement a glorified Else If Statement ... but there are differences as we shall see in the next lecture note page.

If no matching value can be found in any of the case statements, then a **default** section can be processed.    Think of the default as an "else" in an "else if" statement.

**Important:** There are some limitations with the switch, it can't handle **multiple expressions**, but it can run much **faster** than an else if statement, and thus, that is its real value.

## SYNTAX

```
switch (expression)
{

     case value1:
         body;
         break;

     case value2:
         body;
```

```
          break;

      ...
      default:
          one or more statements;
          break;

  }
```

- Notice the colon after each case
- As many cases as necessary may be used
- The body does not have to be in curly braces for multiple statements
- If **break** is not present, the call falls through to the next case rather than exiting the switch.

The **default** is equivalent to the final **else** in a nested **else if**

---

## 4 New Reserved Words

- switch
- case
- default
- break

Important note:

The values used in the case MUST be simple constants or constant expressions. NO variables! NO Character Strings!

We'll learn about character strings in the next few weeks, but think of them as words or numbers between double quotes.

| Case Value | Valid? | Why? |
|---|---|---|
| 5 | Valid | Simple Integer Value |
| 5.02 | Valid | Simple Float Value |
| 'A' | Valid | Simple Character Value |
| "ABC" | Invalid | Character Strings not allowed |
| | | Constant Expression |

| | | |
|---|---|---|
| 5 + 4 | Valid | always evaluates to 9 |
| ABC | Invalid | Variables not allowed |

# Switch Example

Let's **compare and contrast** the **switch** statement with the **else if** statement. Some will argue that a switch might be quicker execution wise if properly designed, and it might be easier to read, but a big advantage that an else if would have is handling a **range of values**. If I wanted to check to see if an exam score was between 90 and 100, I could say:

```
if ( score >= 90 && score <= 100 )
```

with a switch statement I would have to account for every possible score value:

```
case 90:
case 91:
case 92:
case 93:
case 94:
case 95:
case 96:
case 97:
case 98:
case 99:
case 100:
```

As you can see, that can be **problematic** in that its easy to leave out a value and it is a lot more code. However, what if instead of processing a raw test score, it asked the student for their letter grade, such as A, B, C, D, or F. Based on their response, they could go through the series of questions that could potentially improve my bank account. Rewriting the previous else if program example with a **switch** might then look like the code shown below.  Note how each **switch case** has a corresponding **break statement**, which is critical to making the switch work.  Otherwise, processing would just continue with the next **case** statement instead of dropping to the end of the switch.  The corresponding **break** statement associated with the **default case** is optional, but highly recommended so you get into the habit of adding a **break** to each **case** statement.

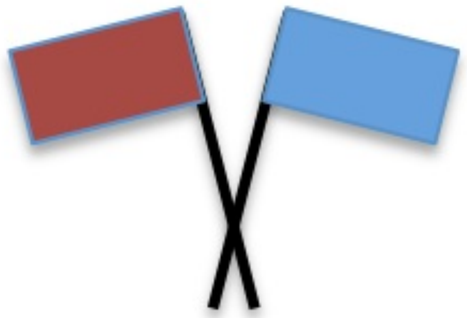
Try it out in IDEOne at: http://ideone.com/xUKhdF ... feel free to put in the grade you want and see what happens :)

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

```
#include <ctype.h> /* for the toupper function */

int main ()
{

char answer [80];  /* string to hold user response */
char grade;           /* your exam grade */

 /* Ask student for their exam grade */
printf ("\n Enter your exam grade: ");
grade = getchar();

 /* convert character to upper case using the C library function toupper */
grade = toupper ( grade );

 /* Skip a few lines */
printf ("\n\n");

 /* Student Self Service based on their grade */
switch ( grade )
{

    case 'A':

       printf ("\nExcellent Job: Grade is an A \n");
       break;

   case 'B':

       printf ("\nGood Job: Grade is a B \n");
       printf ("\nWould you like an A? (y/n): ");
       scanf ("%s", answer);

       /* Come on, answer yes, professor needs a new electronic toy */
       if ( answer [0] == 'y' || answer [0] == 'Y' )
       {
          /* All right, a customer */
         printf ("\n Make check payable to Tim Niesen for $200 \n");
         printf ("... a small price to pay for your education \n");
```

```
      }

      break;

    case 'C':

      printf ("\nFair Job: Grade is a C \n");
      printf ("Would you like an A? (y/n): ");
      scanf ("%s", answer);  /* we'll discuss strings next week, but its a series of characters */

      /* time to make some real money, my favorite students */
      if ( answer [0] == 'y' || answer [0] == 'Y' )
      {
          printf ("\n Make check payable to Tim Niesen for $600 \n");
          printf ("... an excellent price to pay for your education \n");
      }
      else
      {
           printf ("\nWould you like an B? (y/n): ");
           scanf ("%s", answer);

          if ( answer [0] ==  'y'  ||  answer [0] ==  'Y' )
          {
              printf ("\nMake check payable to Tim Niesen for $400 \n");
              printf ("... a great price to pay for your education \n");
          } /* if */
       } /* else */

      break;

   case 'D':

     printf ("\n Poor Job: Grade is a D \n");
     printf ("... You're beyond a bribe to get a better grade \n");

     break;

  case 'F':
     printf ("\n Looks like you Failed. Grade is a F \n");
     break;
 default: /* if it gets here, not a valid grade */
     printf ("\n Invalid/Incomplete Grade ... can not process ...\n");
     break;  /* optional, but recommended */
} /* switch */
return (0);
} /* main */
```

# Flags



A **flag** is a variable which is normally set to TRUE or FALSE, remember that in C, TRUE has a value of 1, and FALSE is 0.

It records that some **event** has taken place.

For example, below is a test to determine if a number is odd or even. If you take an integer number and divide it by two and there is no remainder, then the number is even, otherwise it is odd. For example, take the even number 4 and divide it by 2, and you get 2 Remainder 0. Contrast that with the odd number 5 divided by 2, which yields 2 Remainder 1. In the code example below, we test a number using the modulus operator which will return the remainder of a divide operation. The name of the **flag variable** below is **is_even**.

It is set to 1 (TRUE) if the remainder is 0, signifying that the number is even, otherwise it is set to 0 (FALSE) to indicate it is an odd number.

```
if ( number % 2 == 0 )   /* divide a number by two, is the remainder zero? */
{
   is_even = 1;   /* set flag to True, it is even */
}
else /* odd */
{
   is_even = 0;   /* set flag to False, it must be odd */
}
```

It may be tested later to see if an event happened or not

```
if ( is_even == 1 )   /* these three do */
if ( is_even != 0 )   /* the same */
if ( is_even )         /* thing */

if ( ! is_even )       /* these two do */
if ( is_even == 0 )   /* the same thing */
```

The expression is evaluated and its value is set to 1 if the expression is satisfied (TRUE), to 0 if the expression is not satisfied (FALSE).

**is_even** may be set to 1 (TRUE) or 0 (FALSE) by your program when it is determined that a value is or is not an even number.

Negation reverses the value. If **is_even** was FALSE, then ! **is_even** has a value of 1 and is TRUE. If **is_even** was TRUE (a non-zero value) then negation sets it to 0, which is FALSE.

A flag can also have more than just two values ... can you think of something to model that has more than two states?

# Unary Operator Precedence

Some operators have very high precedence. The **UNARY** operator NOT ( ! ) is an example of one with very high precedence.

```
! ( x < y )
```

Requires parentheses or interpreted as:

```
( ! x ) < y
```

If you run the NOT operator on a TRUE value

```
! ( 22 )
```

... it would return a value of FALSE (or zero), since !(TRUE) is FALSE.

Of course, it can be a variable or expression as well, as its obvious the if would be true if you hard coded a value of 22:

```
if ( ! value)

if ( ! ( value + x + PI ) )
```

Same would happen with negative numbers:

```
! ( -22 )
```

If you negated a zero value, then it would be TRUE, as not FALSE is TRUE

```
! ( 0 )

! ( 0.0 )
```

Remember, only 0 (and 0.0 is really just zero) is FALSE, anything else is TRUE

The best way handle the original statement above is
with parentheses:

```
! ( x < y )
```

equates to

```
x >= y
```

# Conditional Expression Operator

The **conditional expression operator** is a valid operator in C as an alternative to an if else statement. I would caution that many coding standards in industry frown upon its usage and it is easy to code it incorrectly and it can be hard to test if the statement is complex. It is however, widely used in development and deployment of **Macros**, which we will study in Chapter 13, the C Preprocessor.

## SYNTAX

```
condition ? expression1  :  expression2;
```

The **condition** is usually some type of relational expression that will evaluate to either TRUE or FALSE. If the **condition** is TRUE, it will execute the statements in **expression1**, otherwise, if it is FALSE, it will execute the statements in **expression2**.

## EXAMPLES

```
max_value = ( a > b ) ? a : b;

s = ( x < 0 ) ? -1 : x * x;
```

The above two statements are equivalent to:

```
if ( a > b )
    max_value = a;
else
    max_value = b;

if ( x < 0 )
    s = -1;
else
     s = x * x;
```

## Pitfall:

Many coding standards frown upon the use of conditional expressions in industry as they can be harder to read and test depending on their complexity. When in doubt, use the standard **if else** expressions over these **conditional** expressions. However, they are part of the C language, so they should be covered and we will use them again when we discuss macros in Chapter 13.

# Continue Statement Revisited

Last week in our discussion of looping, we covered the **continue** statement. The example I provided wasn't very interesting because continue statements are normally only executed based on some event that is captured by an **if statement**.

Recall that the **continue** statement works similar to the **break** statement.  Unlike the break statement that forces the termination of a loop, the continue allows the next iteration of the loop to execute, skipping any code in between for just that specific loop iteration.

Consider how the continue statement is designed in my sample program below.   Note how once the number in my loop is set to 15, that the **continue** statement is executed within the **if** body. At that point, execution goes to the end of the loop (the while statement at the end), yet the loop continues with its next set of iterations until the number is incremented to 21.   Watch it work in IDEOne at:  http://ideone.com/X2S2JU.

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

    int number = 0; /* simple integer value */

    /* do loop execution */
    do
    {
        if ( number == 15 )
        {
            number += 3;

            continue; /* go directly to end of loop */
        }

        printf ("value of number: %i\n", number);

        number += 3;

    } while( number <= 21 ); /* end of loop */

    return 0;
}
```

The **output** from this program in shown below. Again, note that when the number is set to 15, the body within the if statement will first increment the number by 3 and then execute the **continue** statement. At that point, execution will go to the **end of the loop**, bypassing the last two statements in the loop. Execution continues on with the number being incremented by three and printed until the number 21 was reached.

```
value of number: 0
value of number: 3
value of number: 6
value of number: 9
value of number: 12
value of number: 18
value of number: 21
```

# Variable Declarations within Conditionals (C99)

The **C99 standard** allows for variables to be *declared within conditional statements* such as an if or else construct. The scope of any variable declared in this manner is only known to that particular conditional body, and that's a good thing as it makes your program less prone to errors if that variable location gets corrupted later on.

To show a very simple example, consider the code below which presents a variable named **myVal** being declared followed by an if statement which uses that variable inside its body. This variable can be accessed and used anywhere within its function (such as main).

```
int myVal;  /* declared outside the if statement */

if ( y > 100 )
{
   myVal = y+10;
   /* do other things */
}
```

Now, if the only time you plan to use the variable **myVal** is inside this conditional statement, then it is best declared inside the if statement body as shown below to limit its scope.

```
if ( y > 100 )
{
   int myVal; /* better, scope limited to the if statement */

   myVal = y+10;
   /* do other things */
}
```

The sample program below has two instances of declaring the **myVal** variable: 1) For the main function, and 2) Inside the if body. While it would be better to make each variable name declaration unique (i.e., don't have two variables with the same name), I present this example to show you that **each** myVal variable has its ***own memory address*** so they are separate in scope from each other.

Take some time to try it out at http://ideone.com/vN86Fv where I've added print statements to display the memory address assigned to each myVal variable. I find that printing addresses in *hexadecimal* is much easier to view and work with then dealing with a very large integer number. Note that when you print addresses in your code, you can't guarantee what address in memory the system will assign to a variable each time you run it. However, you can display the memory address of each variable during a specific run, which will verify to us in this example that each myVal variable declaration is assigned its own unique memory address.

```
#include <stdio.h>
int main (void)
{
   printf ("1) Start of main declarations");

   int myVal = 5;  /* belongs to the main function */

   printf ("\n\n=>Address of myVal in main: %lx\n\n", &myVal);

   printf ("2) Let's enter the if statement:\n\n");

   if (1 == 1)   /* will always be true */
   {
      int myVal = 10;  /* not the same as the first declared myVal */

      printf ("=>myVal in the if statement is %i\n", myVal);
      printf ("=>Address of myVal in the if statement: %lx \n", &myVal);

   } /* end if */

   printf ("\n3) After the if statement:\n");

   /* This is one declared before the if else condition */
   printf ("\n=> myVal after the if statement is %i\n", myVal);
   printf ("=>Address of myVal after the if statement: %lx \n", &myVal);

   return (0);

} /* end main */
```

## Sample Output

```
1) Start of main declarations

=>Address of myVal in main: bfc81d28

2) Let's enter the if statement:

=>myVal in the if statement is 10
=>Address of myVal in the if statement: bfc81d2c

3) After the if statement:

=> myVal after if statement is 5
=>Address of myVal after the if statement: bfc81d28
```

Note that in both cases, the initial **myVal** variable declared in the *main function* is not affected by the **myVal** variable with the same name declared in the *if* statement *body*. I have in the past declared variables in a conditional body if I know for sure I will never need to access them anywhere in my program when I leave that particular conditional construct. It gives me piece of mind that I'm efficiently using memory and stack space, and coding with security in mind. We'll cover those latter topics more in detail in a few weeks.

## Switch Statements and C99 Variables?

I tried to see if I could declare variables within a switch statement and the IDEOne compiler would not accept it. The test code I added to the sample program above is shown below:

```
switch (loop_test)
{
   case 0:
      int myVal;
      printf ("\n=>Address of myVal in switch case 0: %lx \n", & myVal);
      break;
   case 1:
      int myVal;
      printf ("\n=>Address of myVal in switch case 1: %lx \n", &myVal);
      break;
   default:
      printf ("No switch case\n");
      break;
} /* end switch */
```

I got this for a syntax error from the IDEOne compiler:

```
prog.c:38:15: error: a label can only be part of a statement and a declaration
is not a statement
   case 1: int myVal;
           ^
prog.c:38:19: error: redeclaration of 'myVal' with no linkage
```

However, if should work if you put an if or if-else statement and associated body within a case statement:

```
switch (loop_test)
{
   case 0:
      if (x > 5)
      {
         int myVal;
         printf ("\n=>Address of myVal in switch case 0: %lx \n", & myVal);
      }
      break;

      /* add other cases */

} /* end switch */
```

# Common Conditional Mistakes

There are many mistakes you can easily make when working with conditional logic in C.  We covered many of them this week, but let's summarize them here given a program that will test and process whether a light switch is turned on or turned off.

Try out the program at:  https://ideone.com/HqPRtx

```
#include <stdio.h>

#define OFF 0
#define ON 1

int main ()
{
    int light_switch = ON; /* a light switch */

    /* Check to see if the lights are on     */
    /* using both an if and switch statement */
    if (light_switch == ON)
        printf ("The Lights are ON\n);

    switch (light_switch)
    {
        case ON:

            printf ("The Lights are ON\n");
            break;

        case OFF:

            printf ("The Lights are OFF\n");
            break;
    } /* switch */

    return 0;
}
```

The **output** from this program is shown below. Because the light_switch was initialized to ON (i.e., 1), it will print that the lights are on given the tests and processing on both the **if** and **switch** statements.

```
    The Lights are ON
    The Lights are ON
```

Let's take a look at all the possible common mistakes you could make in this program using conditional logic statements:

| THE WRONG WAY | THE RIGHT WAY |
|---|---|
| *// **Mistake 1**: Using = instead of ==*<br><br>if (light_switch **=** ON)<br><br>... This will set light_switch **equal** to 1. It is probably the number 1 error new and experienced C and C++ programmers will make. In this case, the if conditional will ALWAYS be true, since the value of 1 is **true**. | *// **Solution:** Use **==** instead of **=***<br><br>if (light_switch **==** ON)<br><br>... This will correctly check whether the light_switch **is equal** to 1, it is asking the question if the light_switch is **equal to 1** or not, with the answer being either **true** or **false**. |
| *// **Mistake 2:** Mixing case for C statements*<br>*If ( light_switch == ON )*<br><br>... this would not compile, you would get the warning below, and then likely a few more compiler errors as it | *// **Solution:** Use "if", C is **case sensitive***<br>*if ( light_switch == ON )*<br><br>... Use "**if**", not "**If**" or "**IF**", as C is case sensitive for |

| | |
|---|---|
| started getting confused later on. `warning: implicit declaration of function ‘If’` | nearly all of the C syntax outside of constants and macros (which we will learn about this later) is **lower case**. |
| // ***Mistake 3:*** *Forgetting to use parentheses* `if light_switch == ON` ... This would not compile, you would get: `error: expected ‘(’ before ‘light_switch’` | // ***Solution: Add parentheses*** *with conditionals* `if ( light_switch = ON )` |
| // ***Mistake 4:*** *Adding an semicolon after* // *the condition* `if (light_switch == ON) ;` `{` `printf ("It is time for bed, turn off the lights);` `light_switch = OFF;` `}` ... This would compile, but never execute the two statements in the body, regardless or whether the if statement evaluated to true or false. | // ***Solution: Don't add semicolons*** *after a* // *conditional statement* `if (light_switch == ON)` `{` `printf ("It is time for bed, turn off the lights);` `light_switch = OFF;` `}` ... Also, do not add semicolons after either of the **curly braces** as well. |
| // ***Mistake 5:*** *Forgetting curly braces for* // *multiple body statements* `if (light_switch == ON)` `printf ("It is time for bed, turn off the lights);` `light_switch = OFF;` ... This would compile, but only the first statement, the printf, would be associated with the if statement. Would effectively run as: | // ***Solution:*** *Always **enclose body** of two* // *more statements **within curly*** // ***braces** to indicate start and end* // *of the body* `if (light_switch == ON)` `{` |

```
if (light_switch == ON)
{
   printf ("It is time for bed, turn off the lights);

}

light_switch = OFF;
```

```
   printf ("It is time for bed, turn off the lights);
   light_switch = OFF;
}
```

```
//  Mistake 6:  Forgetting the break
//                    statement with each case

switch (light_switch)
{
   case ON:
       printf("The Lights are ON\n");

   case OFF:
       printf("The Lights is OFF\n");

} /* switch */
```

... Without a **break** statement in the **case** sections, if the light_switch had a value of ON (i.e., 1), then it would process the statements in the case area for ON, then continue down and process the next **case** area (OFF), thus printing:

```
The Lights are ON
The Lights are OFF
```

```
// Solution:  Add a break statement at the end
//                   of each case */

switch (light_switch)
{
   case ON:
       printf("The Lights are ON\n");
       break;

   case OFF:
       printf("The Lights are OFF\n");
       break;

} /* switch */
```

... With a **break** statement now inserted in each **case** area, if the light_switch had a value of ON (i.e., 1), then it would process the statements in the case area for O*N*, then go to the end of the **switch**, thus printing:

```
The Lights are ON
```

## Summary

This week we learned about conditional processing. The essence of good computer programming is to factor into your design the ability to handle all possible conditions that could take place. You need to continuously ask yourself: "What could happen next?".

We reviewed concepts within the C programming language that are common to computer languages, such as: if statements, else, else ifs, nested ifs, switches, flags, continues, breaks, and the conditional expression operator. We also learned about the concept of true and false, as well as boolean logic, and how it all works within a relational expression.

## For Next Week

For next week, do homework assignment #2 and try some of the exercises from the *Conditionals* chapter you read this week. I'll post my answers to some of them next week. Just take your homework 1 program as a starting point, incorporate any of my grading comments (if any), and add a new variable to capture **overtime**, which is defined as any hours worked in a given week that were greater than 40 hours. Those overtime hours will be paid at **time and a half**.

For example, if you worked 45 hours and made $10 an hour, the first 40 hours would be paid at an hourly **wage** of $10 and the 5 **overtime hours** (those hours over 40 hours) would be paid at an hourly **wage** of $15 (1.5 x $10). The homework will also ask you to start incorporating **constants** going forward (we reviewed them last week).

Additionally, as always, test your knowledge this week with my short **Quiz**.

We will discuss **Arrays** next week, going far beyond what the book has to offer. For you **Star Trekking** fans out there, you will find that Mr. Scoot has a problem with the way Mr. Sili designed the **Shields** that protect Mr. Scoot's precious Star Ship. Could using Arrays offer the right solution? Will Mr. Scoot find that option acceptable? Find out next week, same class time, same class channel.

# HOMEWORK 2
# MAKING DECISIONS

Write a C program that will calculate the gross pay of a set of employees.

Unlike the first assignment, you don't have to prompt for the number of employees to process up front.   Instead use a constant as your loop test.

For each employee the program should prompt the user to enter the clock number, wage rate, and number of hours as shown below.

The program determines both the overtime hours (anything over 40 hours) and the gross pay, and outputs the following format. Hours over 40 are paid at time and a half.

```
----------------------------------------------
Clock# Wage Hours OT Gross
----------------------------------------------
098401 10.60 51.0 11.0 598.90
```

In the example above, since the employee worked 51 hours, 11 of those hours are overtime and are paid at time and a half. The first 40 hours are paid at the standard time using the hourly wage rate. If no overtime hours are worked, just indicate 0.0 hours in your output.

***Example:*** If your hourly wage was $10 an hour, and you worked 46 hours, the first 40 hours would be your regular pay and paid at $10 an hour for a total of $400. Your overtime hours would be any hours past 40 hours, which in this case is 6 hours. To figure out the overtime pay, each of those 6 hours would paid at time and half, which would be $15 and hour (to calculate: 1.5 * $10 = $15). In this example, your standard pay would be $400 and your overtime pay would be $90 for a total gross pay of $490.

At a minimum, I would recommend at least adding a new variable to store your overtime hours. If you wish, you could optionally create variables for overtime_pay and standard_pay, that would be up to you, but you don't need to print these values.

Column alignment, leading zeros in Clock#, and zero suppression in float fields are important.

With this and in future assignments, please start using constants. For example, instead of hard coding a value for 40.0, use a constant:

```
    #define STD_HOURS 40.0
```

Replace 40.0 in your code with the symbolic constant STD_HOURS

There are a few other constants you should define and use for this assignment.

***Use at least one more constant in your homework***

Use the following data as test input:

```
  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
```

Do not use any material from any chapters we have not yet covered (such as adding your own functions, arrays, structures, pointers ... we'll get to them in the future), unless we have discussed them in the lectures notes or on-line discussions.

Start with your graded homework 1 assignment and modify the code as needed for this homework.