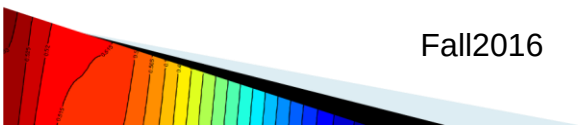
GNG 1106

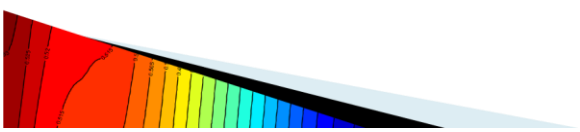
Fundamentals of Engineering Computation

Module 5 - Decisions and Loops



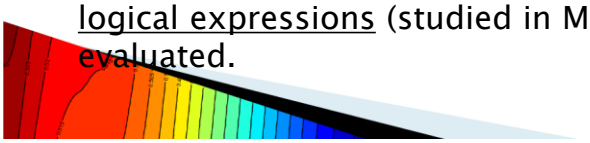
Fall2016

Topic 1: Decision Instructions



Decision Instructions

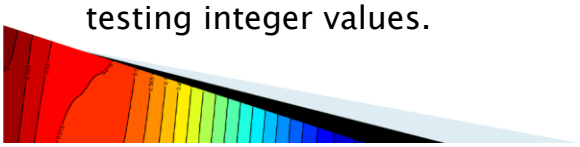
- Recall that computers execute instructions one at a time
  - In fact C instructions are broken down into smaller machine instructions
  - A program can be view as a list of instructions to be executed by the CPU
  - Would be nice to have the CPU make decisions on executing certain instructions
- The decision instructions
  - Computers can skip instructions, more precisely jump to instructions based on a value of TRUE or FALSE (in reality a value of 1 or 0)
  - From our point of view, decisions are taken once logical expressions (studied in Module 2) have been evaluated.



3

Decision instructions in C

- Three decision instructions exist in C
- The `if/else`
  - The basic decision instruction with an instruction block to be executed if a logical expression is TRUE and an optional instruction block if the logical expression is FALSE.
- The `Else-if`
  - The short-hand form multiply nested decision instructions
- The `switch` (optional use in this course)
  - An alternate form of the `Else-if` instruction when testing integer values.



4

The if Decision Instruction

- The `if` decision instruction allows us to isolate C instructions and to have the program execute these instructions only if a logical expression is true.
- Syntax to isolate a single C instruction:

```
if (logical expression)
  instruction;
```

Careful: there is no `;` here.

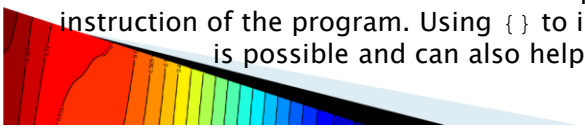
The C instruction directly below the `if ()` is executed only if `logical expression` evaluates to true.

- Syntax to isolate many C instructions:

```
if (logical expression)
{
  instruction 1;
  instruction 2;
   ⋮
  instruction n;
}
```

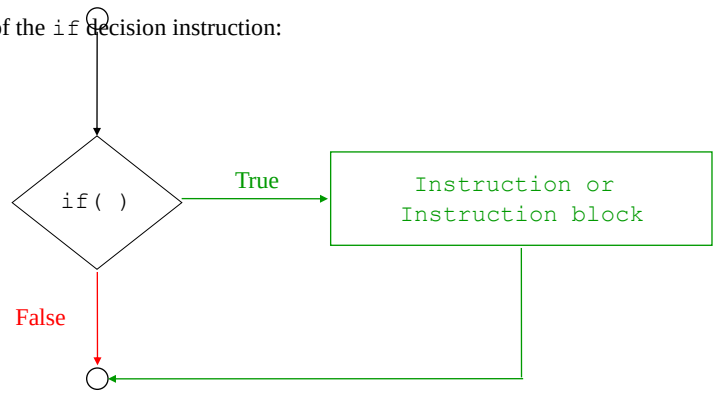
The C instructions delimited by a pair of `{}` directly below the `if ()` are executed only if `logical expression` evaluates to true.

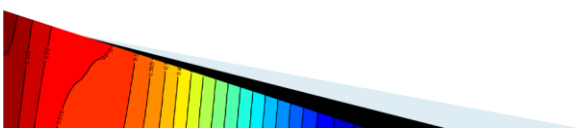
- It is very good practice (and mandatory in this course) to indent the instructions under the `if` in order to help visualize the logical instruction of the program. Using `{}` to isolate a single instruction is possible and can also help clarify the code.



5

- A **flowchart** is graphical description of a complex language instruction, algorithm or small program.
  - A flowchart is a convenient way to the visualize logical flow.
  - Lozenges are used to represent decisions and simple instructions (bloc) are shown as boxes.
- Flowchart of the `if` decision instruction:





6

- Using the `if` instruction to isolate one instruction:

```
if (a < 10.0)
  a = a + 1.0;
```

  - or to isolate an instruction block:

```
if (a < 10.0)
{
  a = a + 1.0;
  b = b + 1.0;
}
```

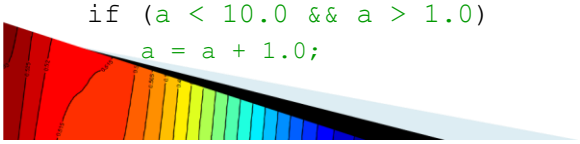
If `a` was declared as:
`double a = 5.5;`
then in all of these examples the logical expression evaluates to true and the instructions isolated by the `if`'s are executed.

- We can also have nested `if`'s:

```
if (a > 1.0)
  if (a < 10.0)
    a = a + 1.0;
```

  - The above can also be written using logical operators:

```
if (a < 10.0 && a > 1.0)
    a = a + 1.0;
```



7

- Using the `if` instruction to isolate one instruction:

```
if (a > 10.0)
  a = a + 1.0;
```

  - or to isolate many instructions:

```
if (a > 10.0)
{
  a = a + 1.0;
  b = b + 1.0;
}
```

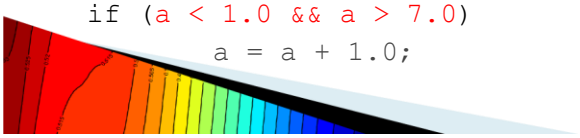
If `a` was declared as:
`double a = 5.5;`
then in all of these examples the logical expression evaluates to false and the instructions isolated by the `if`'s are not executed.

- … nested `if`'s:

```
if (a < 1.0)
  if (a > 7.0)
    a = a + 1.0;
```

  - the above written using logical operators:

```
if (a < 1.0 && a > 7.0)
        a = a + 1.0;
```



8

if/else Decision Instruction

- The `if/else` instruction allows us to specify a set of instructions to be executed if the logical expression is true and another set of instructions if the logical expression is false.
- Syntax to isolate single C instructions:

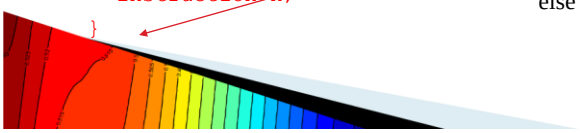
```
if (logical expression)
  instruction 1;
else
  instruction 2;
```

Careful: there is no ; here.

- Syntax to isolate multiple C instructions:

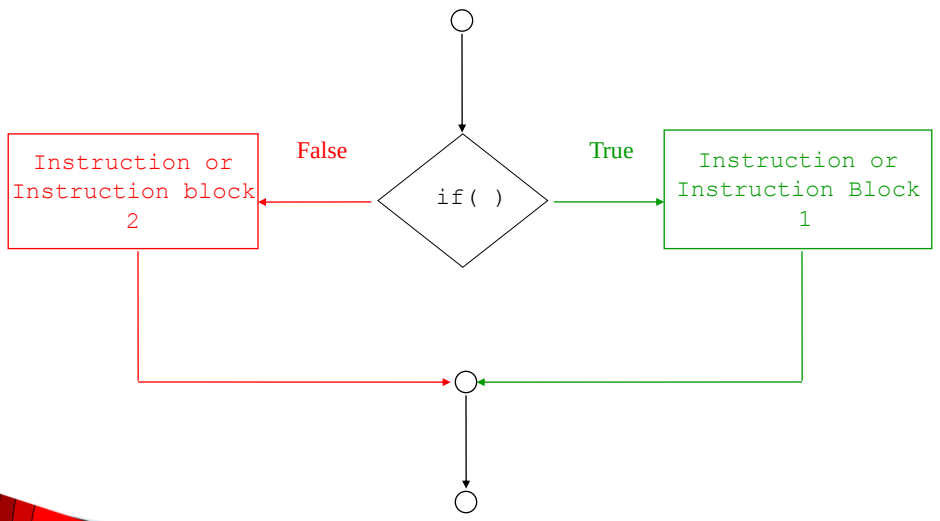
```
if (logical expression) {
  instruction 1;
  instruction n;
}
else {
  instruction 1;
  instruction n;
}
```

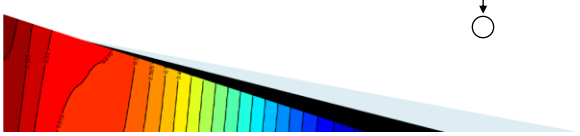
Note the difference in the arrangement of the pair of {} in this case; this is also acceptable to the compiler and still clearly delimits the scope of the if and else parts.



9

- Flowchart of the `if/else` decision instruction:





10

- Example:

```
if (grade >= 90)
  printf("You get an A+.\n");
else
{
  printf("Less than 90.\n");
  printf("Less than A+.\n");
}
```

→ Executed if grade is greater than or equal to 90.

→ Executed if grade is less than 90.

- We can also introduce other `if/else` instructions into an `if/else` instruction:

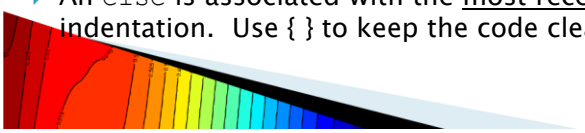
```
if (x > y)
  if (y < z)
    k = k + 1;
  else
    m = m + 1;
else
  j = j + 1;
```

→ Executed if x>y and y<z

→ Executed if x>y and y≥z

→ Executed if x≤y

- An `else` is associated with the most recent `if`, regardless of indentation. Use { } to keep the code clear.



11

- The use of `{}` is highly recommended in a complex logic instruction to clarify the intended scope of the `if`'s and `else`'s and to help avoid programmer logic errors.
  - Consider the following instruction with misleading indentation:

```
if (x > y)
  if (y < z)
        k = k + 1;
else
    j = j + 1;
```

→ Executed if x>y and y<z.

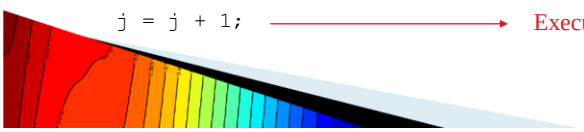
→ Executed if x>y and y≥z; not if x≤y as was probably intended by the programmer. Recall: an `else` is always associated with the most recent `if`.

  - What the programmer probably wanted is:

```
if (x > y)
{
  if (y < z)
      k = k + 1;
}
else
  j = j + 1;
```

→ Executed if x>y and y<z.

→ Executed if x≤y.

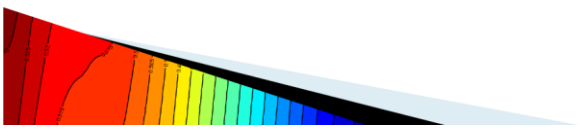


12

- The following decision instruction is commonly encountered and is created from many levels of nested `if/else`'s:

```
if (choice == 1)
  printf("First choice selected.\n");
else if (choice == 2)
  printf("Second choice selected.\n");
else if (choice == 3)
  printf("Third choice selected.\n");
else
  printf("Wrong selection!\n");
```

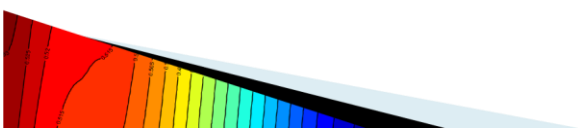
  - The variable choice is compared with the integer values 1 to 3.
  - If a logical expression is true (say the first one) then the message that follows is printed to the screen (`First choice selected.`) and execution will continue with the instruction following the instruction; i.e.: after `printf("Wrong selection!\n");`
  - The last `else` (`Wrong selection!`) is executed only if none of the `if`'s are true.
  - The above is such a popular decision instruction that programmers rarely add indentation and `{}`. Exceptionally, this is not considered to be in bad programming style.



- Note however, how indentation and `{}` clarifies the intended logic in the previous example:

```
if (choice == 1)
  printf("First choice selected.\n");
else
{
  if (choice == 2)
    printf("Second choice selected.\n");
  else
  {
      if (choice == 3)
          printf("Third choice selected.\n");
      else
          printf("Wrong selection!\n");
  }
}
```

  - The above executes in exactly the same way.



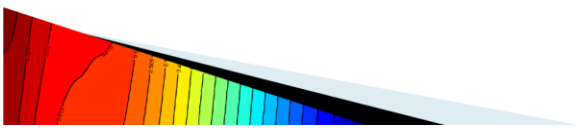
- Careful with the logic in nested `if/else` instructions.
  - Let's say that we have the following decision instruction:

```
if (grade >= 90)
  printf("You have an A+\n");
else if (grade >= 85)
  printf("You have an A\n");
```

  - The above works correctly; the variation shown below however, does not.

```
if (grade >= 85)
  printf("You have an A\n");
else if (grade >= 90)
  printf(You have an A+\n");
```

  - See the problem? What happens if `grade` is equal to `90`?



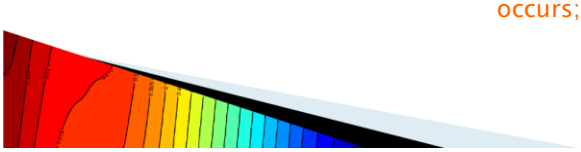
Caution: Using = and = =

- Careful when using the assignment operator = and the equality operator ==; mixing then up may not cause a compilation error but will of course cause an execution error that can be more difficult to detect.
  - Consider the following commonly encountered error:

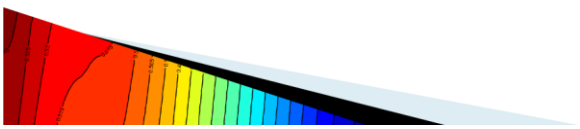
Intended code	Erroneous code
`if (a == 2)` `  printf("…\n");`	`if (a = 2)` `  printf("…\n");`
True only when `a` is equal to `2`.	Always true since the logical result of an assignment is always true.

  - The following is another popular mistake:

Intended code	Erroneous code
`a = 2;`	`a == 2;`
Assigns the integer `2` to the variable `a`.	Logical expression is evaluated during execution but no assignment occurs; `a` remains unchanged.

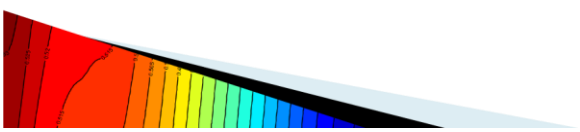


Topic 2: Loop Instructions



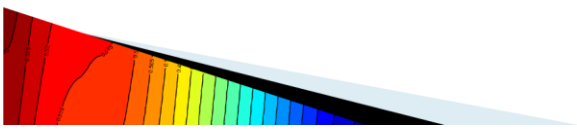
Loop Instructions

- Most programs require repetition instructions in order to execute one or more instructions a certain number of times.
- A loop can be:
  - a definite repetition loop if we know in advance the number of repetitions, or
  - an indefinite repetition loop if we do not know in advance the number of repetitions.
- The number of repetitions,
  - in a definite repetition loop is controlled by a counter (usually an integer variable),
  - whereas the number of repetitions in an indefinite repetition loop is controlled by a sentinel (usually an integer variable).



Loop Instructions in C

- There exists in C three repetition instructions:
  - the `while` instruction,
  - the `do/while` instruction, and
  - the `for` instruction.



`while` Repetition Instruction

- Allows the execution of one or more C instructions while a logical expression remains true.
- Syntax for repeating a single C instruction:

```
while (logical expression)
   instruction;
```

Careful: there is no ; here.

The C instruction directly below the `while ()` is executed as long as `logical expression` remains true.

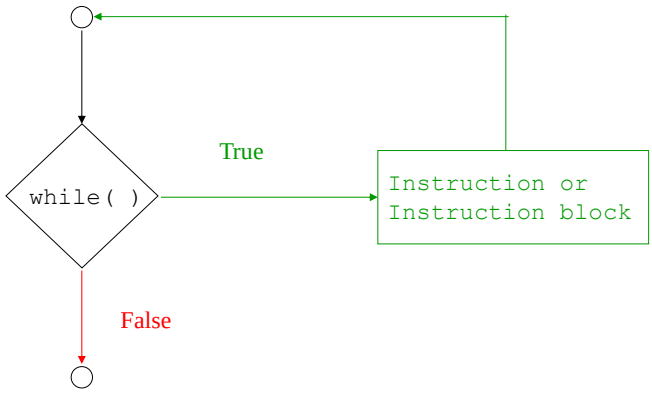
- Syntax for repeating multiple C instructions:

```
while (logical expression)
{
   instruction 1;
   instruction 2;
   ⋮
   instruction n;
}
```

The C instruction block directly below the `while ()` is executed as long as `logical expression` remains true.

- The `while` instruction is a pre-tested loop.
  ➔ It is possible that the instructions in the loop are never executed.
- It is very good practice (and mandatory in this course) to indent the instructions in the instruction block of a `while` loop in order to help visualize the instruction of the program. Using `{}` to include a single instruction in a `while` loop is possible and can also help clarify the code.

- Flowchart of the `while` instruction for repeating a single instruction:



- Operation of a `while` instruction in a program:

```
⋮
while (logical expression)
{
    instruction 1;
    instruction 2;
}
⋮
```

True

False

Program execution flows to the `while` instruction and the `logical expression` is tested. If true the instructions between the { } are executed and the `logical expression` is again tested. If false, program execution continues after the }

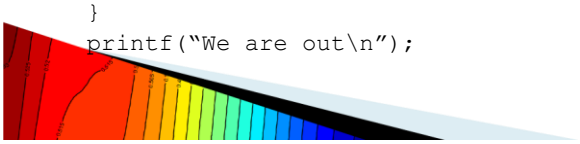
- The instructions in a `while` instruction must modify at least one variable in the logical expression; otherwise, an infinite loop results.
- When an infinite loop occurs, the user must usually interrupt the program. The mechanism for program interruption is machine dependent.
  - PC's running Windows: Simultaneously press the keys `Ctrl Alt Del` to invoke the task manager, select the task to end (`YourProgName.exe`) and click on End Task.
  - Simply close the "cmd" Window that is executing the program.
  - PC's and Workstations running UNIX: type `Ctrl c` simultaneously in the window where the program is executing.

- Example of a definite repetition loop using the `while` instruction:

```
int ctr;
ctr = 1;
while (ctr <= 10)
{
     printf("Iteration number: %d\n", ctr);
     ctr = ctr + 1;
}
printf("ctr upon exit of the loop: %d\n",ctr);
```

- Example of an indefinite repetition loop using the `while` instruction:

```
int sentinel;
sentinel = 1;
while (sentinel != 0)
{
     printf("Enter 0 to exit the loop\n");
     scanf("%d", &sentinel);
}
printf("We are out\n");
```



`do/while` Repetition Instruction

- The `do/while` repetition instruction is similar to the `while` instruction except that it is a post-tested loop.
  - ➔ The instructions in the loop are always executed at least once.
- Syntax for repeating a single C instruction:

```
do
   instruction;
while (logical expression);
```

- Syntax for repeating many C instructions:

```
do
{
   instruction 1;
   instruction 2;
   ⋮
   instruction n;
}
while (logical expression);
```

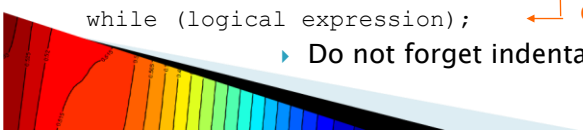
Careful: there is no `;` here.

The C instruction between the `do` and `while ()` is repeated as long as `logical expression` remains true.

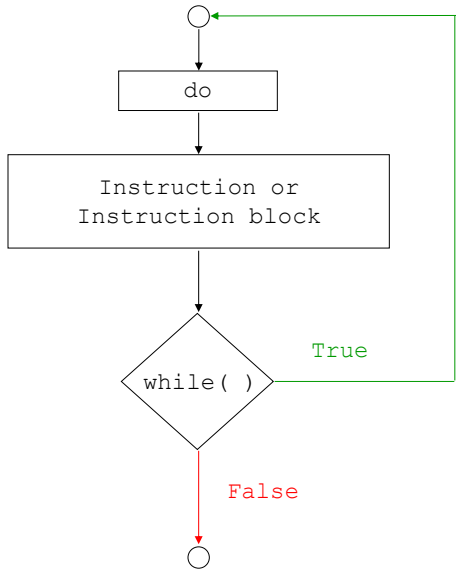
The C instruction block between the `do` and `while ()` is repeated as long as `logical expression` remains true.

Careful: there is a `;` here.

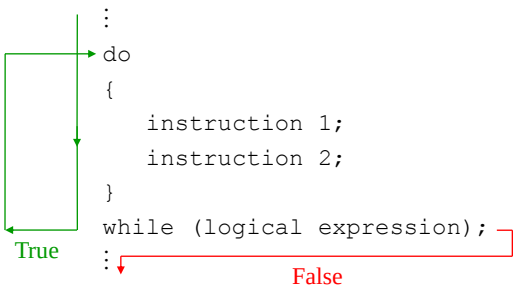
- Do not forget indentation.



- Flowchart of the `do/while` instruction for repeating a single instruction:



- Operation of a `do/while` instruction in a program:



Program execution flows into the `do/while` instruction, performing at least once the instructions between the `{}`. The `logical expression` is then tested and if true, execution returns to the `do` statement and the instructions between the `{}` are repeated. When `logical expression` becomes false, program execution continues after the `while ();`

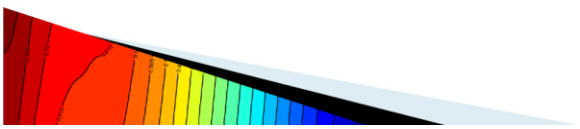
- The instructions in a `do/while` must modify at least one variable in the logical expression; otherwise, an infinite loop results.
- It is possible to break out of an infinite `do/while` loop in the same manner as in the case of an infinite `while` loop.

- Example of a definite repetition loop using the `do/while` instruction:

```
int ctr;
ctr = 1;
do
{
   printf("Iteration %d\n", ctr);
   ctr = ctr + 1;
}
while (ctr <= 10);
```

- Example of an indefinite repetition loop using the `do/while` instruction:

```
int sentinel;
sentinel = 1;  // Is this line required?
do
{
   printf("Enter 0 to exit the loop\n");
   scanf("%d", &sentinel);
}
while (sentinel != 0);
```



`for` Repetition Instruction

- The `for` looping instruction is a definite repetition instruction that makes use of a counter.
- Syntax to repeat a single C instruction:

```
for (expression 1; expression 2; expression 3)
   instruction;
```

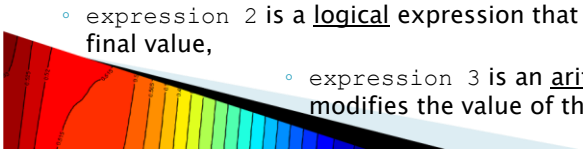
Careful: there is no ; here.

- Syntax to repeat a multiple C instructions:

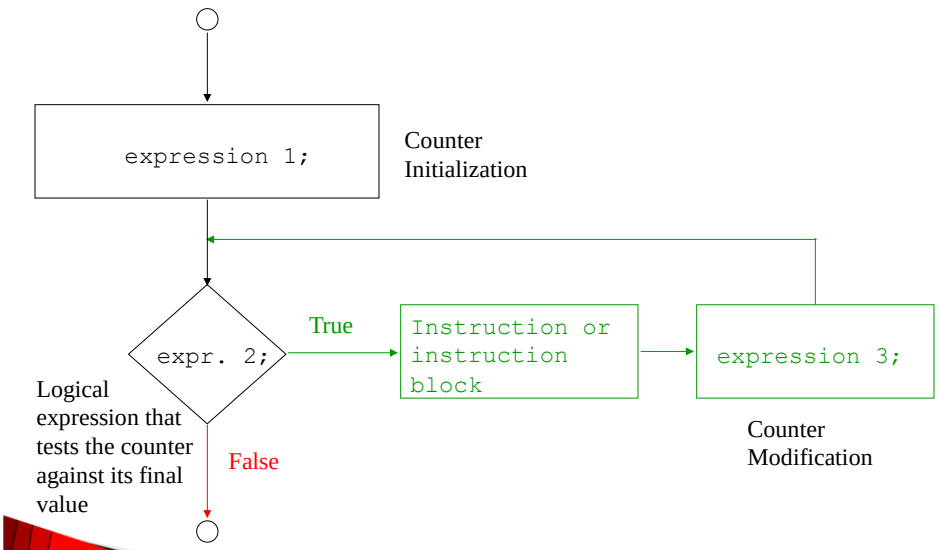
```
for (expression 1; expression 2; expression 3)
{
   instruction 1;
   instruction 2;
   ⋮
   instruction n;
}
```

Careful: there are ; here.

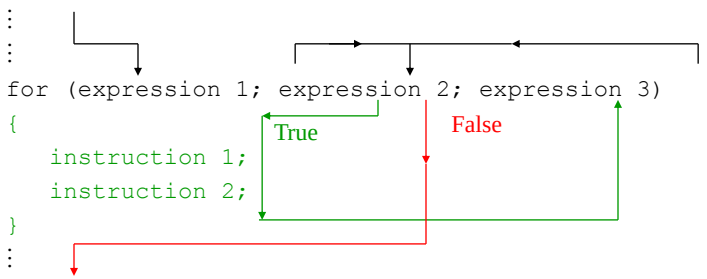
- The three expressions in the `for` loop have the following role:
  - `expression 1` is an arithmetic expression that initializes the counter,
  - `expression 2` is a logical expression that tests the counter against its final value,
  - `expression 3` is an arithmetic expression that modifies the value of the counter.



- Flowchart of the `for` repetition instruction for repeating a single instruction:



Operation of a `for` instruction in a program:



A Program execution flows first into `expression 1` which is evaluated <u>once</u> to initialize the loop counter.

B `expression 2` is then evaluated; recall that it is a logical expression containing the loop counter;

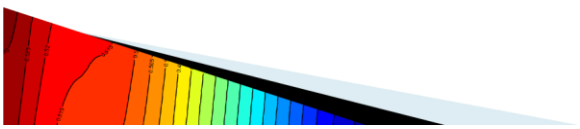
D if `expression 2` evaluates to false, then the instructions between the `{}` are skipped and the program continues to execute after the loop;

E if `expression 2` evaluates to true, then the instructions between the `{}` are executed and `expression 3` is evaluated to modify the value of the counter;

F back to step B.

- The `for` instruction is a pre-tested loop.
  ➔It is possible that the instructions in the loop are never executed.
- It is very good practice (and mandatory for this course) to indent the instructions in the loop in order to help visualize the structure of the program. Using `{}` to include a single instruction in a `for` loop is possible and can also help clarify the code.
- Example of a repetition loop using a `for` instruction:

```
int ctr;
for (ctr=1; ctr<=10; ctr = ctr + 1)
    printf("Iteration %d\n", ctr);
```



31

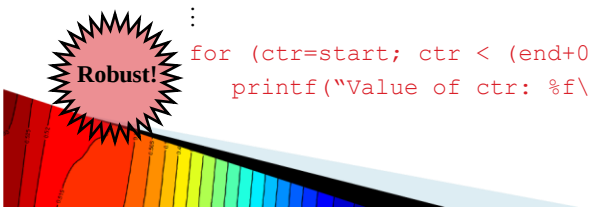
On the Use of Real Variables as Loop Counters

- It is possible to use a real variable as a counter to control looping in a definite repetition loop.
- In theory, real variables can be tested for equality against a given value but in practice this is not robust due to the imprecise nature of arithmetic with real variables on computers.
- It is best to modify the logical expression for the loop termination condition such that testing for equality is not required.
  - Looping from 1.0 to 2.5 inclusively might be coded as:

```
double start=1.0, end=2.5, inc=0.5, ctr;
⋮
for (ctr=start; ctr <= end; ctr=ctr+inc)
   printf("Value of ctr: %f\n", ctr);
```

  - The above loop is best implemented as:

```
⋮
for (ctr=start; ctr < (end+0.5*inc); ctr=ctr+inc)
   printf("Value of ctr: %f\n", ctr);
```



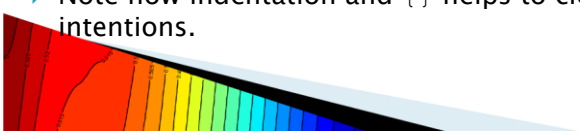
32

Nested Loops

- Any C instruction can be placed inside any of our repetition instructions; this includes decision instructions and other loops. Loops placed inside of loops are called nested loops.
- Suppose for example that we want to compute z points for the surface $z = x^2 + y^2$ over the range $-10 \le x \le 10$ and $-10 \le y \le 10$; we could do this using nested `for` loops:

```
double x,y,
for(x = -10.0; x < 11.0; x = x + 2.0)
{
   for(y = -10.0; y < 11.0; y = y + 2.0)
   {
          z = x*x + y*y;
          printf("z = %f\n", z);
   }
}
```

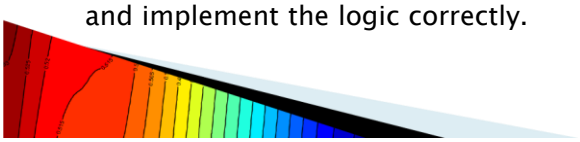
- Note how indentation and `{}` helps to clear up the programmer's intentions.



On the Use of goto's

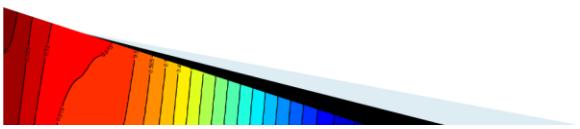
- USE THEM AND GO DIRECTLY TO JAIL!
- A `goto` instruction exists in C.
- Structured program development implies `goto`-less programming.
  - It has been proven that all algorithms can be implemented without using `goto`'s.
  - `goto`'s quickly render a complex program untraceable by humans.
- On a related note:
  - Do not `break;` or `return` to break out of looping or decision instructions! These are forms of goto's.
  - Use the loop control mechanisms and implement the logic correctly.

JACKSON POLLACK'S MOTHER
'Programs with many **goto***s are so tangled and difficult to trace through…'*



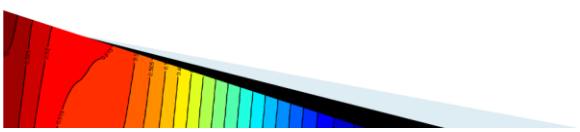
Algorithms: Using Decision and Loop Instructions

- Decision and loop instructions play an important role in developing software algorithms.
  - An algorithm is a description of the steps that a computer must take to complete the desired tasks.
- We shall present a few simple algorithms to demonstrate the use of decision and loop instructions in solving problems
  - These algorithms are basic and can be applied in finding many solutions to problems.



Examples

- Using Decision/Loops for Input
  - selectComputation.c
- Calculating π using accumulation of values in a variable (see next slide)
- Traversing 1D Arrays
  - Finding the minimum value in an array
    - findMin.c
- Traversing 2D Arrays
  - Scanning a matrix to see if it diagonally dominant
  - See 2 slides down.



Example: Calculating π

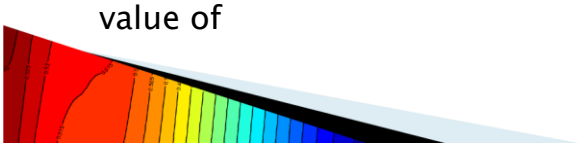
- The value of π using the Gregory-Liebniz series

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \frac{4}{13} + \ldots$$

- The above series can be expressed as

$$\pi = \sum_{i=1}^{N/2} \frac{4}{4*i-3} - \frac{4}{4*i-1}$$

- Where N defines the number of terms used to compute π (if N is odd then one term is dropped)
- Can you see how you can use i as a counter in a loop to compute the value of π.
- The program computePi.c prompts the user for the value of N and calls a function to compute the value of

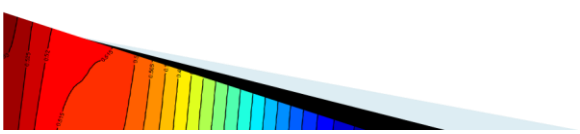


Example: Is matrix Diagonally Dominant?

- A square matrix is Diagonally Dominant when for each row *i* in the matrix, the diagonal element a_{ii} is larger than the sum of all other elements (i.e. non-diagonal) in the row. The following matrix is diagonally dominant.

$$M = \begin{bmatrix} 15 & 9 & 4 \\ 7 & 11 & 3 \\ 6 & 1 & 8 \end{bmatrix}$$

- The program isDiagDominant.c gives a program that contains a function that evaluates if a matrix is diagonally dominant.



Next Module

- Topic 1: File I/O Library
- Topic 2: Plotting Library