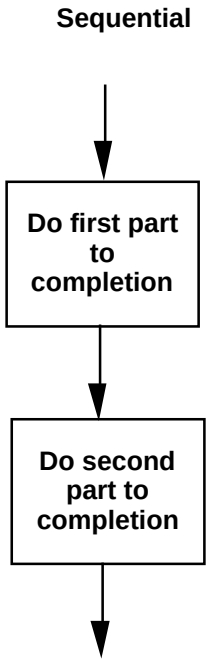
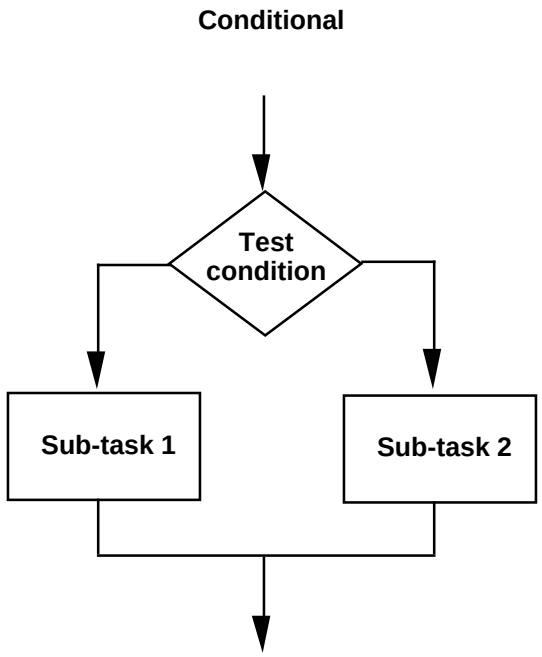
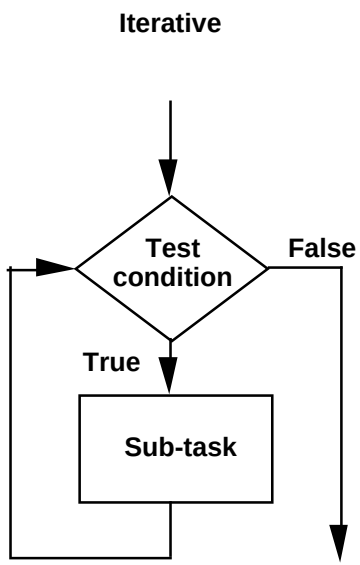
Program Control Flow - Iteration
2/23/01 Lecture #8 16.070

Basic Constructs of Structured Programming







Clarification: The *switch* Statement

- Once a case is matched in a switch statement, all subsequent cases will be executed, **including the default case!**
- Driving directions to the airport

```
switch (location)
  {
   case MIT: walk_to_Kendall();
   case kendall: board_redline();
   case redline: switch_greenline();
   case greenline:
switch_blue_line();
break;
   default:  ask_directions();
  }
```

Program Control Flow - Iteration

- Iteration constructs repeat a sequence of code in a controlled manner
- Iteration directs the computer to perform the same set of operations over and over until a specified condition is met
- Three C statements for looping
  - *while*
  - *for*
  - *do … while*

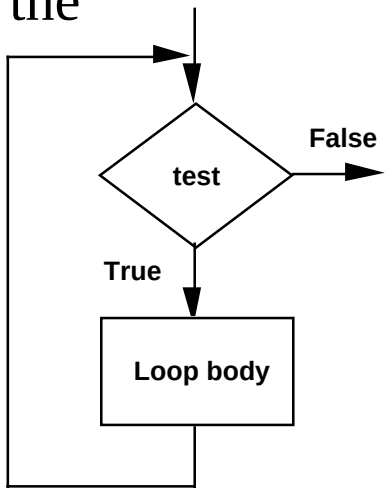
Iteration - The *while* Statement

- Repeatedly executes a statement while a test condition (an expression) evaluates to true

```
while (<expression>)
   statement;
```

  - Test condition is checked before each cycle, or iteration, through the loop
  - If expression evaluates to TRUE (non-zero), statement is executed (again)

- Pretest: expression is tested before each execution of the statement



- Use brackets to group multiple statements

```
while (<expression>)
{
   statement1;
   statement2;
}
/* end while */
```

The *while* Statement Template

- Recommended approach to using while

  get first value to be tested
  while the test is successful
      process value
      get next value

- Note that the body includes something that **changes the value** of the test expression. Why? What happens if value being tested doesn't change?

```
int variable = 1;
while (variable == 1)
{
   statement1;
update value of variable;
   statement3;
}
/* end while */
```

The *while* Statement for Counter Controlled Loops

- *while* can be used for counter-controlled loops
  - Declare loop control variable
  - Assign initial value to the variable
  - Test loop control variable by comparing to a final value
  - Update loop control variable: increment/decrement by a certain value
  - E.g., This loop iterates while the value of x is less than 10.

```
int x = 0;
while (x < 10)
 {
   printf ("%d ", x);
   x = x + 1;
}
/* end while */
```

  - Produces the following output:

```
0 1 2 3 4 5 6 7 8 9
```

The *while* Statement for Sentinel Controlled Loops

- *while* can be used for sentinel-controlled loops
  - Declare sentinel variable and decide on termination value
  - Initialize sentinel variable
  - Use sentinel variable in loop control expression
  - Change value of sentinel variable so that loop is eventually exited
  - Example: code to compute the square of a number entered via keyboard

```
int number;
printf ("Enter an integer to square; enter zero to stop: ");
scanf ("%d", &number);
while (number)
 {
  printf ("The square of %d is:  %d\n", number, number *
number);
printf ("Enter an integer to square; enter zero to stop: ");
scanf ("%d", &number);
}
/* end while */
```

The *while* Statement - Initialization

- Caution: Always be sure that the variable being checked in the *while* test has been initialized!
  - In example above, omit the *printf* and *scanf* lines prior to *while* statement. What would be the outcome?

```
int number;
while (number)
{
printf ("Enter an integer to square; enter zero to stop: ");
scanf ("%d", &number);
   printf ("The square of %d is:  %d\n", number, number *
number);
}
/* end while */
```

The *while* Statement - Termination

- Common mistake when using while -- loop termination
  - Mistakes in body can cause an infinite loop, causing program to never terminate

```
int x = 0;
while (x < 10)
   printf ("%d ", x);
/* end while */
```

  - Mistakes in test condition can cause an infinite loop, causing program to never terminate

```
int x = 0;
while (x > -10)
{
   printf ("%d ", x);
x = x + 1;
}
/* end while */
```

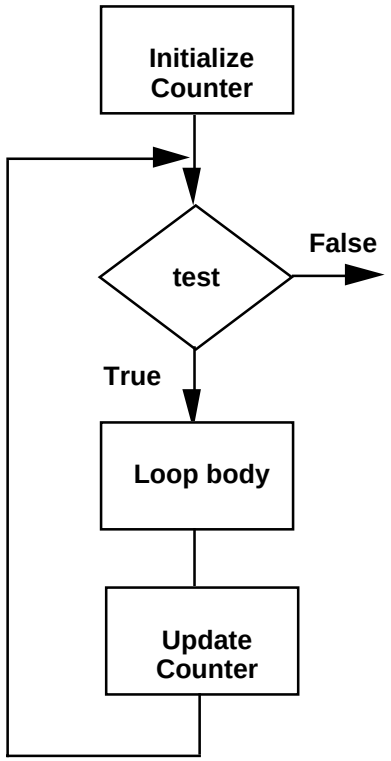
  - Make sure test value changes, and changes in right direction!

Iteration - The *for* Statement

- The *for* statement is designed as shorthand for looping with the following conditions
  - When you need to initialize one or more variables before entering the loop
  - When you need to change the value of one or more variables each time through the loop
- Most frequently used of all iterative statements

```
for (<initialize>; <test>; <update>)
    loop_body;
```

- Combines three actions into one
  - Initialize: Initialize counter
  - Test: Compare counter to limiting value
  - Update: Increment counter each time through the loop



Execution of the *for* Statement

- Initialize: Initialization is performed just once before the first iteration, but is always performed regardless of test result
- Test: <test> expression gets evaluated before every iteration to determine if another iteration should be executed
- Update: <update> expression is evaluated at the **end** of every iteration. Used to prepare for the next iteration
- Loop body: Defines the work to be performed in each iteration

```
int num;
int x = 5;
for (num = 5; num < x; num++)
{                  /* begin loop body */
statement1;
statement2;
}                  /* end loop body */
/* end for */
printf ("After loop, num = %d\n", num);
```

Iteration: *for* vs *while*

- Compare the *for* statement to the *while* statement
  - The *for* statement

```
for (<init_exp>; <test_exp>; <update_exp>)
statement1;
/* end for */
```

---------- is equivalent to ----------

  - The *while* statement

```
<init_exp>;
while (<test_exp>)
{
   statement1;
   <update_exp>;
}
/* end while */
```

Recommended Uses of *for* vs *while*

- The *while* statement used for sentinel-controlled loops; where number of repetitions depends on value of variable being tested
- The *for* statement used for counter-controlled: perform "n" number of repetitions
- Note: The *for* statement provides some level of reliability:
  - Compiler will not let you forget an "initialize" expression (although it can be a null statement)
  - Compiler will not let you forget an "update" expression (although it can be a null statement)

Flexibility of *for*

- Decrement operator to count down instead of up

```
int secs;
for (sec = 5; sec > 0; sec--)
printf ("%d seconds!\n", secs);
/* end for */
printf ("We have ignition!\n");
```

- Count by twos, threes, or any number you define

```
int num;
for (num = 2; num < 60; num = num + 2)
printf ("%d \n", num);
/* end for */
```

- Test condition can be other than for the number of iterations

```
/* test for num squared < 100 */
int num;
for (num = 1; num * num < 100; num++)
printf ("%d \n", num);
/* end for */
```

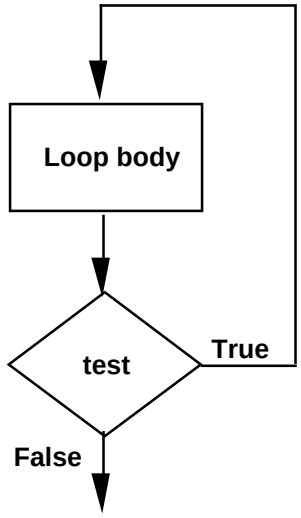
Potential for Errors using *for* Loops

- When using iteration, a common programming error is to iterate through a loop the wrong number of times
- Often, off-by-one iteration due to use of wrong relational operator (e.g., <= vs <)

```
/*  compute factorial:   n! = 1*2*…*(n-1)*n      */
int i;
int factorial = 1;
for (i = 1; i<n; i++)
{
   factorial = factorial * i;
}
/* end for */
```

Iteration - The *do - while* Statement

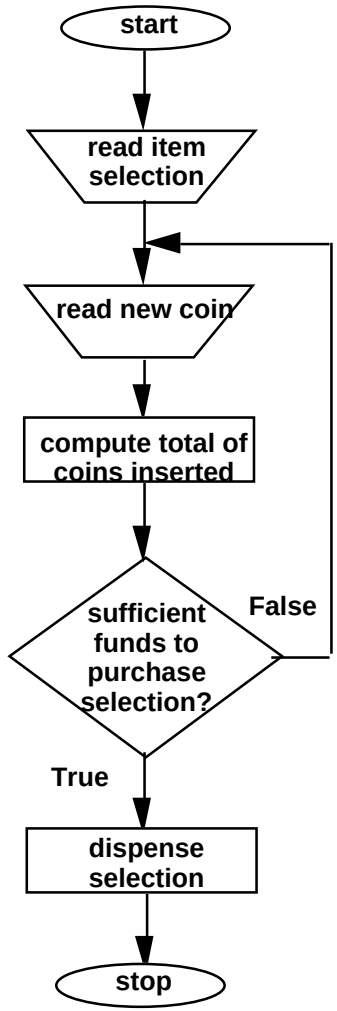
- Only looping structure that performs a Post-test -- tests at the end of the loop
- Loop is executed before the loop control expression is tested.
- After first execution of the loop body, loop control expression is evaluated
  - If loop control expression evaluates to TRUE, loop body is executed again
  - If loop control expression evaluates to FALSE, loop is exited




```
do
<loop body>
while (<test>);
/* end do while */
```

Iteration - The *do … while* Statement

- A *for* loop or a *while* loop can execute zero iterations. A *do-while* loop always performs at least one iteration.
- When would you use this loop? When you know without a doubt that you want to execute loop body at least once, regardless of test condition
- Example: software used in a vending machine that determines if sufficient funds have been inserted to pay for the selected item



Nested Loops

- A loop within a loop construct
- Inner loop is nested within outer loop
- Inner loop must finish before outer loops can resume iterating

Compute the average grade for each student in 16.070

```
for (student = 1; student <= 83; student = student + 1)
{
      for (grade = 1; grade < 10; grade = grade + 1)
      {
            <compute: average = (grade1 + grade2 + … + grade 9) /9 >
      }
      /* end inner for */
      /*  printout average for student  */
      printf ("Student # %d has an average grade of %d. \n", student, average);
}
/* end outer for */
```

Which Loop to Use

- First, decide if you need a loop
- If you need a loop, decide whether you need a pretest or a posttest loop
  - In general, use a pretest loop. Better to look before you leap (loop)
  - Program easier to read if loop test is at beginning of loop
  - Often, loop should be skipped if condition is not met
- A *for* loop is appropriate when loop involves initializing and updating a variable (counter-controlled loops)
- A *while* loop is better when the conditions are otherwise, such as checking for a certain input from the keyboard (sentinel loops)

Infinite Loops in Embedded Systems

- Embedded Systems almost always contain an infinite loop
  - Fundamental difference between embedded systems and programs written for other computer platforms
  - Infinite Loop typically surrounds significant part of program's functionality
  - Necessary because embedded software's job is never done
  - Intended to be run until the world ends or the board is reset, whichever happens first
- For embedded systems, if the software stops running, the hardware is rendered useless

```
while (1)
{
   statement1;
   statement2;
}
/* end while */
```

```
int ever = 1;
for (;ever;)
{
   statement1;
}
/* end for */
```

The *goto* Statement

- The *goto* statement enables program control to jump to another part of the program
- *goto* Statements are controversial
  - Necessary in rudimentary languages such as assembly language
  - Use in high-level languages (e.g., C) frowned upon
  - Tends to produce "spaghetti code"
  - Breaks down the structure of structured programming

Review

- Loops provide a powerful tool to perform iteration
- *while*, *for* are pre-test. Condition must be true for body to execute
- *do … while* is post-test. Body will always execute at least once, regardless of test condition
- Next week
  - Looking inside the computer
  - Read R2.1 and C5.9-5.13
- Reminder: PS3 is on the web and is due 2/28