# Branching and Iteration

- The C language provides three types of decision-making constructs: if-else, the *conditional expression* ?:, and the switch statement.
- It also provides three looping constructs: while, do-while, and for.

# Boolean Expressions

- Evaluate to true or false
- Forms
  - Relational expression: <expr> <relational operator> <expr>
    - Examples:

```
7 < 5
a + b > 6
```

  - Logical expression: <Boolean expr> <logical operator> <Boolean expr>
    - Examples:

```
(x < 7) && (y > 3)
```

# Relational Operators

| Standard Algebraic Relational Operator | C Relational Operator | C Condition Example | Meaning of C Condition |
|---|---|---|---|
| **Inequality** | | | |
| < | < | x < y | **x** is less than **y** |
| ≤ | <= | x <= y | **x** is less than or equal to **y** |
| > | > | x > y | **x** is greater than **y** |
| ≥ | >= | x >= y | **x** is greater than or equal to **y** |
| **Equality** | | | |
| = | == | `x == y` | **x** is equal to **y** |
| ≠ | != | `x != y` | **x** is not equal to **y** |

4th: Ch 4 p. 46
3rd: Ch 5 p. 46

# Logical Operators (Compound Relationals)

- **`&&`** (logical **`AND`**)
  - Returns **`true`** if both conditions are **`true`**
- **`||`** (logical **`OR`**)
  - Returns **`true`** if either of its conditions is **`true`**
- **`!`** (logical **`NOT`**, logical negation)
  - Is a unary operator, only takes one operand following
  - Reverses the truth/falsity of its condition
  - Returns **`true`** when its condition is **`false`**

# Logical Operators Truth Table

| P | Q | P && Q | P \|\| Q | !P |
|---|---|---|---|---|
| true | true | true | true | false |
| true | false | false | true | false |
| false | true | false | true | true |
| false | false | false | false | true |

# Precedence of Operators

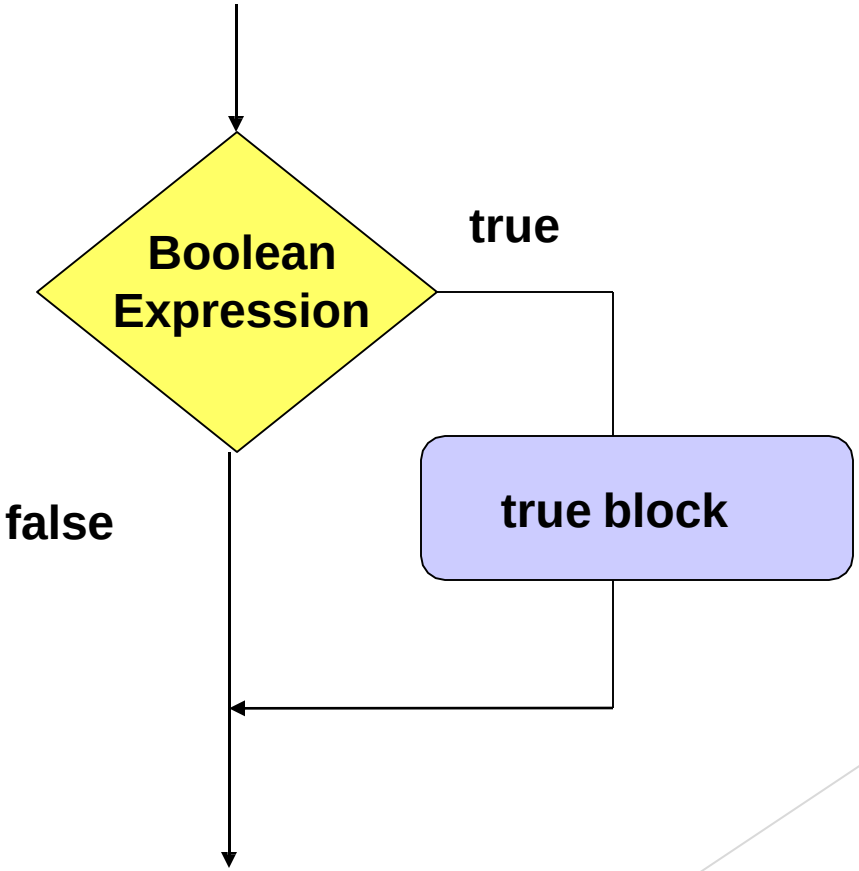
1. (), []
2. Unary +, unary -, !, ++, --
3. Type casting
4. * , / , %
5. + , -
6. <, <=, >, >=
7. ==, !=
8. &&
9. ||
10. =

# The `if` Selection Structure

- Selection structure
  - used when we want the computer to choose between two alternative courses of action

# The *if* Selection Structure

- *if* Statement



# The *if* Selection Structure

- General form of *if:*

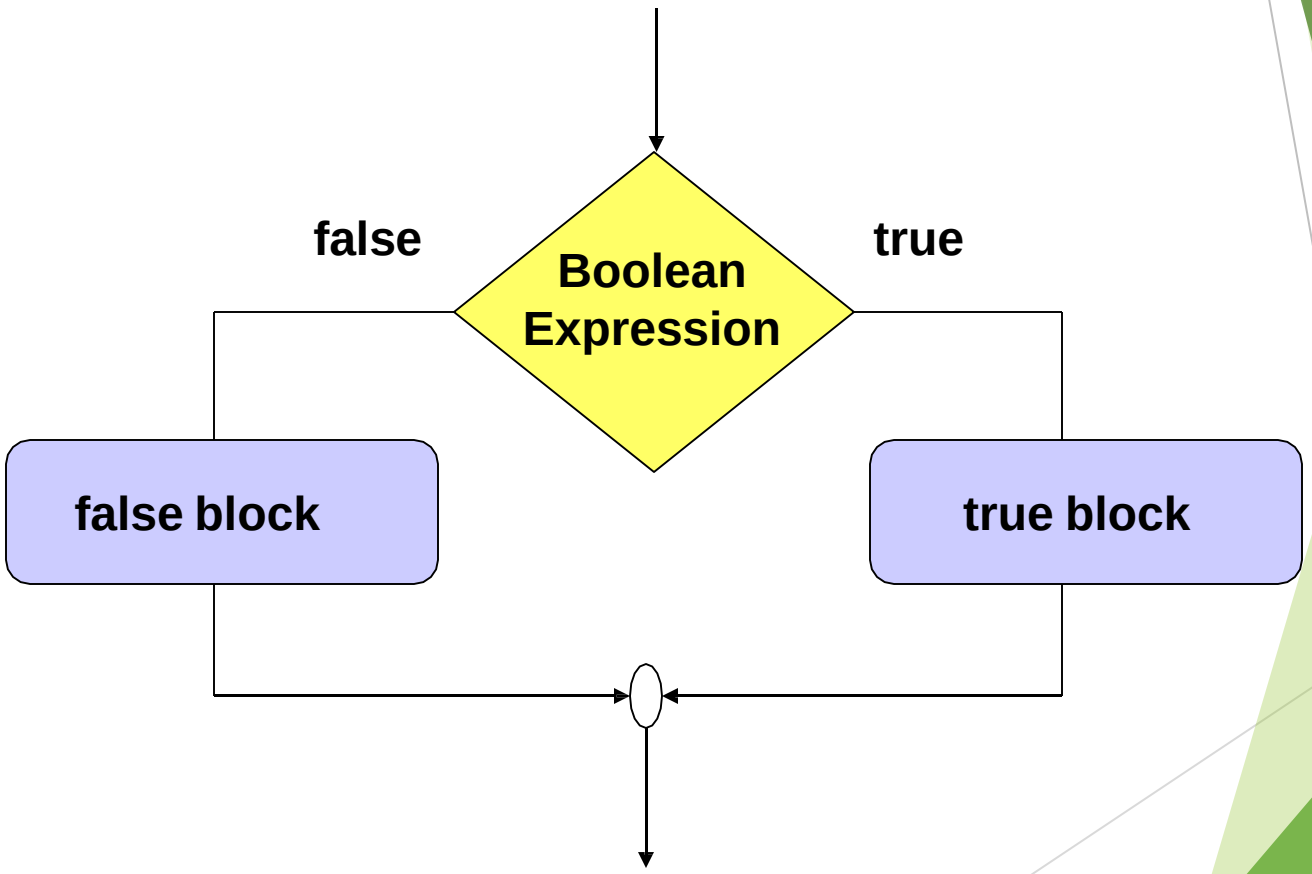
```
if (Boolean Expression)
    {          statement2;
    } statement1;
```

# The `if-else` Selection Structure

- *`if`*
  - Only performs an action if the condition is true
- *`if-else`*
  - A different action is performed when condition is true and when condition is false

# `if-else` Selection Structure

*if-else* statement



# The *if-else* Selection Structure

General form of *if-else*:

```
if (expression)
{
        statement1A;
        statement2A;
        ...
}
else
{
        statement1B;
        statement2B;
        ...
}
```

# The `if-else` Selection Structure

- Nested *`if-else`* structures
  - Test for multiple cases by placing **`if-else`** selection structures inside **`if-else`** selection structures.

# Nested `if-else` Structures

```
if (score >= 70)
{
    if (age < 13)
    {
        printf("Great job\n");
    }
    else
    {
        printf("You passed\n");
    }
}
else
{
    printf("You did not pass\n");
}
```

# The *if-else-if* Construct

```
if (grade >= 90)
   printf("A\n");
else
   if (grade >= 80)
      printf("B\n");
   else
      if (grade >= 70)
         printf("C\n");
      else
         if (grade >= 60)
            printf("D\n");
         else
            printf("F\n");
```


- Once a condition is met, the rest of the statements are skipped

# The *if-else-if* Construct

The standard way to indent the previous code is

```
if (grade >= 90)
   printf("A\n");
else if (grade >= 80
   printf("B\n");
else if (grade >= 70)
   printf("C\n");
else if (grade >= 60)
   printf("D\n");
else
   printf("F\n");
```

# The *if-else* Selection Structure

- Compound statement:
  - Set of statements within a pair of braces
  - Example:

```
if (grade >= 90) {
   printf("Congratulations!\n");
   printf("You made an A this course\n);
}
```

# The *if-else* Selection Structure

–Without the braces, only one statement is executed.

e.g. given the following code:

```
if (grade >= 90)
   printf("Congratulations!\n");
   printf("You made an A this course\n);
```

- The statement,

```
printf("You made an A this course\n);
```

will be executed independent of the value of grade.

- The statement,

```
printf("Congratulations!\n");
```

will execute only if grade is greater than or equal to 90.

# The *dangling* else

```
if (x < y)
   if (x < z)
      printf("Hello\n");
else
   printf("Goodbye\n);
```

**Note:** the compiler matches an else with the closest unmatched if

The above will be treated as

```
if (x < y)
   if (x < z)
      printf("Hello\n");
   else
      printf("Goodbye\n);
```

# The *dangling* else

If the else is to match the outer if, use braces.

```
if (x < y)
{
   if (x < z)
      printf("Hello\n");
}
else
   printf("Goodbye\n);
```

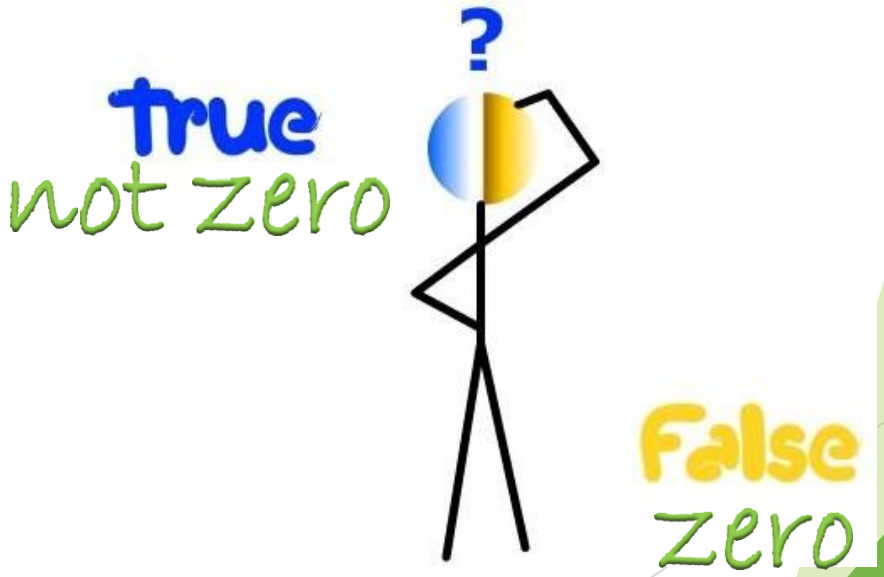
# *if-else* Construct

- To avoid confusion, and possible errors, it is best to use braces even for single statements.
  - However, code will be longer

```
if (x < y)
{
    if (x < z)
    {
        printf("Hello\n");
    }
}
else
{
    printf("Goodbye\n);
}
```

# Conditionals

- C uses an integer to represent Boolean values
  - Zero is interpreted as false
  - Any other integer value is interpreted as true



# Conditionals

- `if (n = 0)` is not a syntax error in C.
  - The expression, *n = 0*, assigns zero to n and the value of the expression is 0. Zero is interpreted as false, and the false branch of the if statement will be taken.
- `if (n = 5)` is not a syntax error in C.
  - The expression assigns 5 to n. 5 is interpreted as true, and the true branch of the if statement will be taken.

```
warning: suggest parentheses around assignment used as truth value
```

# Conditionals

- Remember to use the == operator to test for equality.
- To help catch the error when the equality check involves a constant, put the constant on the left hand side of the ==.
  - For example, use `if (0 == n)`

    instead of `if (n == 0)`

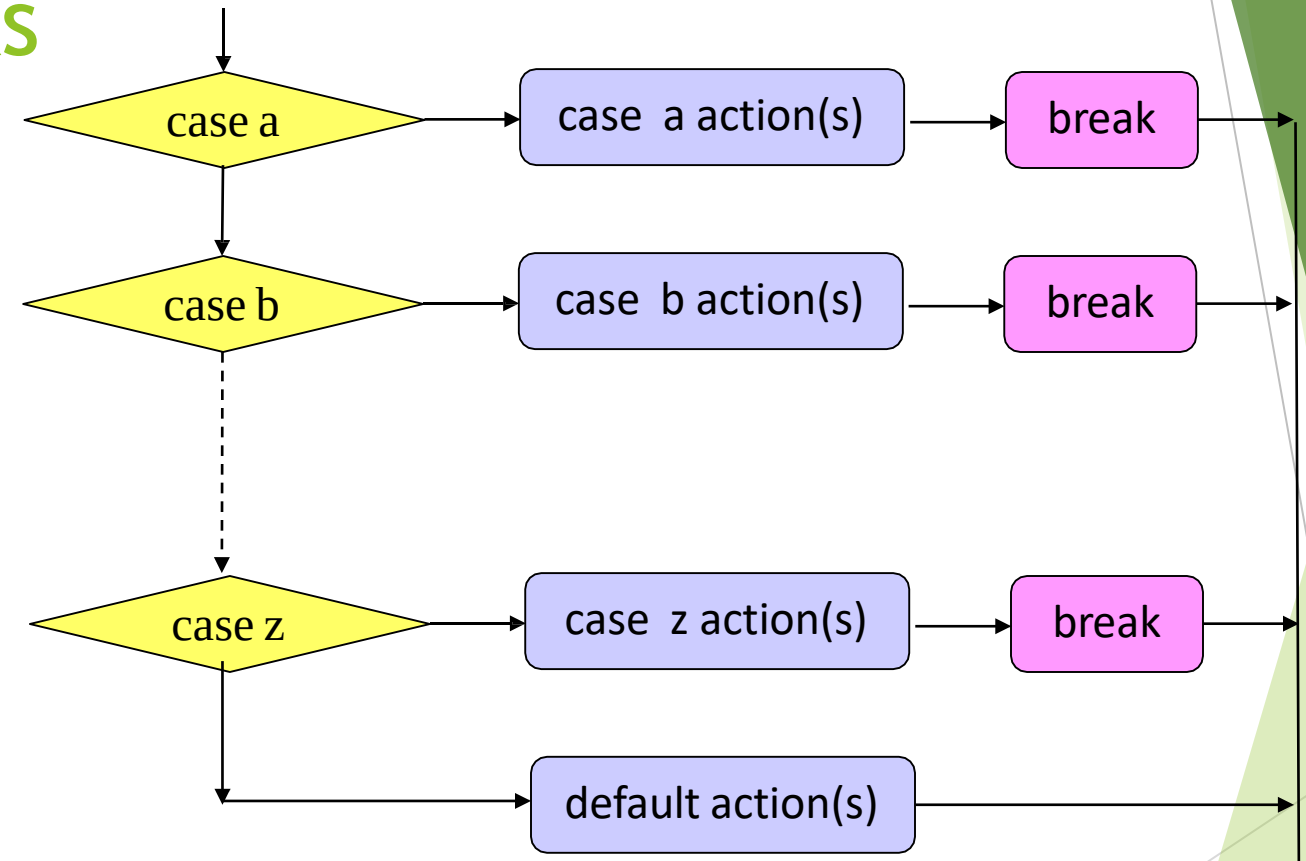
    Since `0 = n` is not a valid assignment in C, the compiler will detect this error when == is intended.

```
error: invalid lvalue in assignment
```

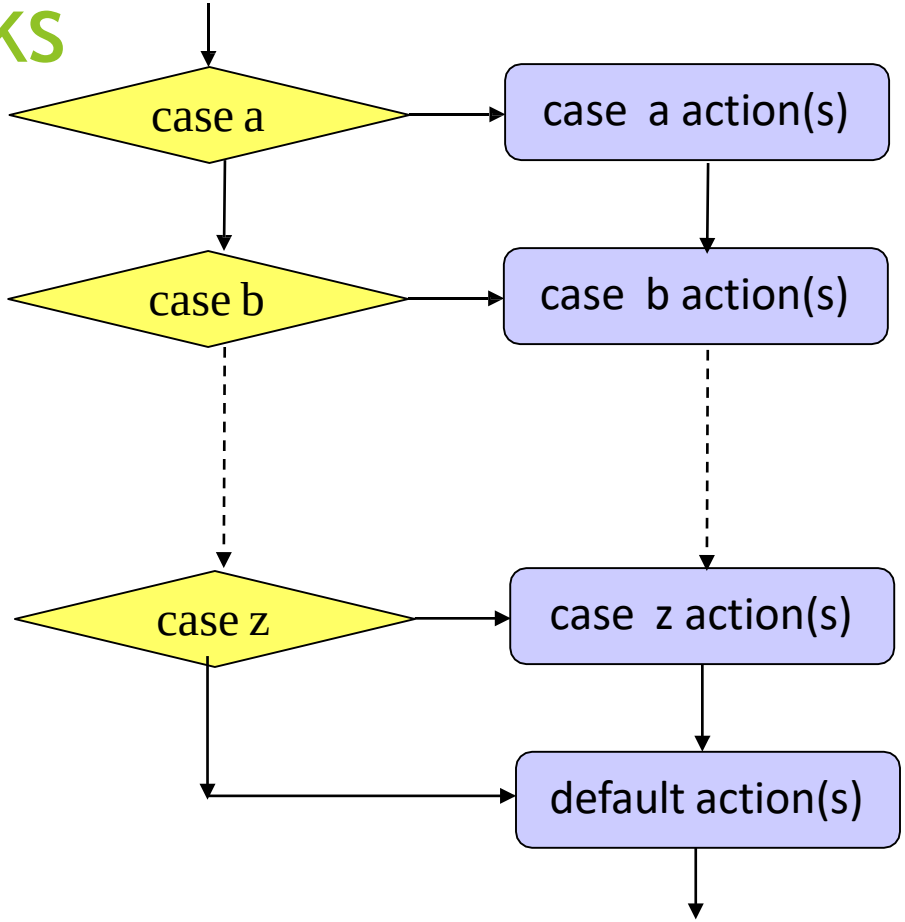
# The *switch* Multiple-Selection Structure

- *switch*
  - Useful when variable or expression is tested for multiple values
  - Consists of a series of **case** labels and an optional **default** case

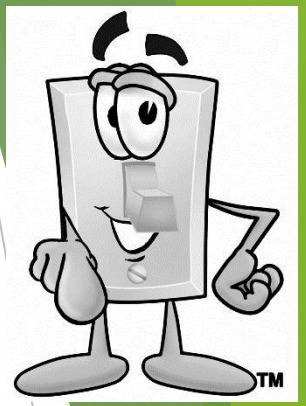
# The *switch* Multiple-Selection Structure With Breaks



# The *switch* Multiple-Selection Structure Without Breaks



# *switch* Statement Syntax



```
switch (switch_expression)
{
        case constant1:
                statementSequence1
                break;
        case constant2:
                statementSequence2
                break;
                                ...
        case constantN:
                statementSequenceN
                break;
        default:
                defaultStmtSequence
}
```

# `switch` Statement

- The switch_expression is compared against the values *constant1, constant2, …, constantN*
  - *constant1, constant2, …, constantN* must be simple constants or constant expressions.
    - Can be a char or an int
    - Best to use the same type constant as the switch expression
      - If not, a type conversion will be done.

# `switch` Statement Reminder


- The `switch` statement ends
  - break statement
  - end of the switch statement
- When executing the statements after a case label, it continues to execute until it reaches a break statement or the end of the switch.
- If you omit the break statements, then after executing the code for one case, the computer will continue to execute the code for the next case.

# Example of *switch*

```
// Accept letter grade and print corresponding points
printf("Enter letter grade: ");
scanf("%c", &letter_grade);
switch (letter_grade)  {
   case 'A':
   case 'a':
      points = 4.0;
      break;
   case 'B':
   case 'b':
      points = 3.0;
      break;
   case 'C':
   case 'c':
      points = 2.0;
      break;
   case 'D':
   case 'd':
      points = 1.0;
      break;
   case 'F':
   case 'f':
      points = 0.0;
      break;
   default:
      points = 0.0;
      printf("Invalid letter grade\n");
}
```

# Quize

- Write C Program to accept degree of student and print his grade according to :
- Lower than 50  fail
- from 50 to 65  pass
- 65 to 75  good
- 75 to 85  very good
- Greater than 85 excellent

# Quize

- Write C Program to design calculator that accept two operands and operator than print the output of arithmetic operation using switch statement.

# Quize

- Write C Program to accept three numbers and print the largest one on the screen.

# Quize

- Write C Program to accept twonumbers and swap between them. Print values of two numbers before and after swap process.

# Programming in C

## Repetition/Looping

Repetition Repetition
Repetition Repetition
Repetition Repetition
Repetition Repetition ...

# Example 1

```
// Read two integers and print sum
int num1, num2, sum;

scanf("%d %d", &num1, &num2);
sum = num1 + num2;
printf("%d + %d = %d\n", num1, num2, sum);
```

- What if we want to process three different pairs of integers?

# Example 2

- One solution is to copy and paste the necessary lines of code. Consider the following modification:

```
scanf("%d %d", &num1, &num2);
sum = num1 + num2;
printf("%d + %d = %d\n", num1, num2, sum);

scanf("%d %d", &num1, &num2);
sum = num1 + num2;
printf("%d + %d = %d\n", num1, num2, sum);

scanf("%d %d", &num1, &num2);
sum = num1 + num2;
printf("%d + %d = %d\n", num1, num2, sum);
```

- What if you wanted to process four sets? Five? Six? ….

# Processing an arbitrary number of pairs

- We might be willing to copy and paste to process a small number of pairs of integers but
- How about 1,000,000 pairs of integers?
- The solution lies in mechanisms used to control the flow of execution
- In particular, the solution lies in the constructs that allow us to instruct the computer to perform a task repetitively

# Repetition (Looping)

- Use looping when you want to execute a block of code several times
  - Block of code = Body of loop
- C provides three types of loops


*while* statement
- *Most flexible*
- *No 'restrictions'*


*for* statement
- *Natural* 'counting' loop


*do-while* statement
- *Always executes body at least once*

# The while Repetition Structure

- Repetition structure
  - Programmer specifies
    - Condition under which actions will be executed
    - Actions to be repeated
  - Psuedocode

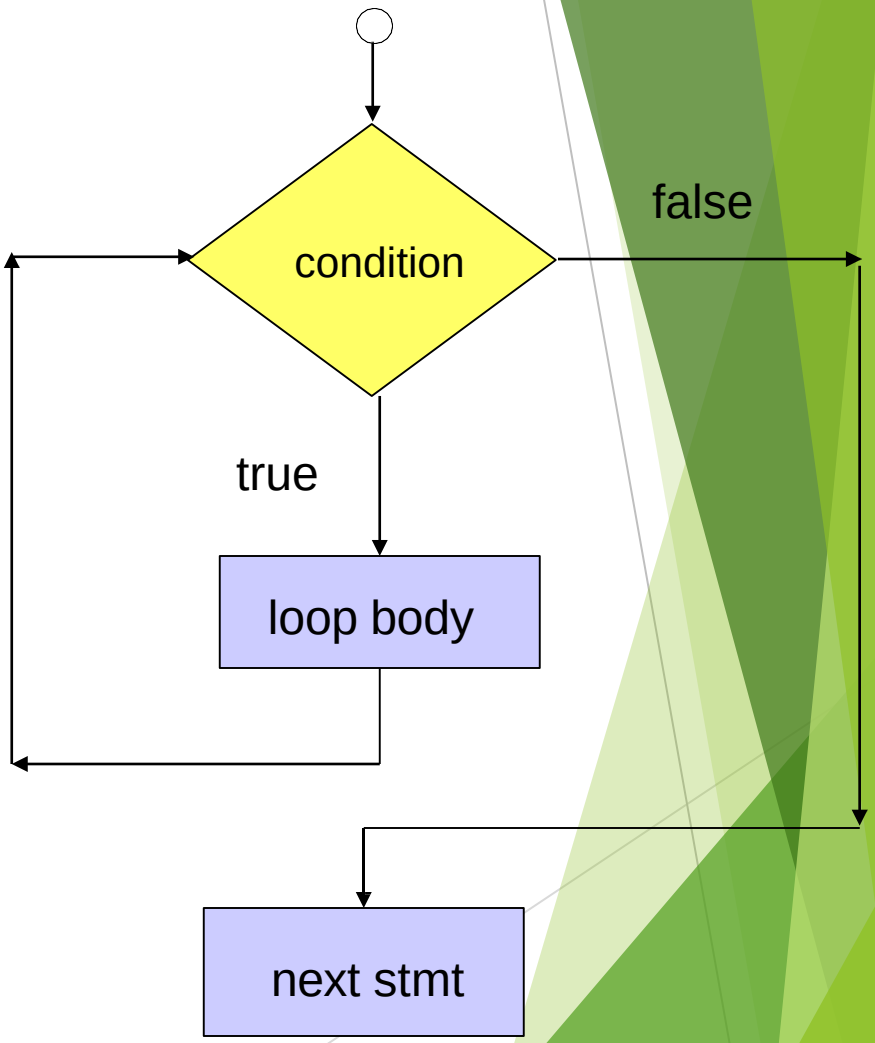
    *While there are more items on my shopping list*

    *Purchase next item and cross it off my list*


- **`while`** loop repeated
  - As long as condition is true
  - Until condition becomes false

# The while Repetition Structure

- The condition is tested
- If the condition is true, the loop body is executed and the condition is retested.
- When the condition is false, the loop is exited.




# The while Repetition Structure

- Syntax:

```
while (expression)
      basic block
```

- Expression = Condition to be tested
  - Resolves to true or false
- Basic Block = Loop Body
  - Reminder – Basic Block:
    - Single statement or
    - Multiple statements enclosed in braces

# Loop Control Variable (LCV)

- The loop control variable is the variable whose value controls loop repetition.
- For a while loop to execute properly, the loop control variable must be
  - declared
  - initialized
  - tested
  - updated in the body of the loop in such a way that the expression/condition will become false
    - If not we will have an endless or infinite loop

# Counter-Controlled Repetition

- Requires:
  1. Counter variable , LCV, initialized to beginning value
  2. Condition that tests for the final value of the counter (i.e., whether looping should continue)
  3. Constant increment (or decrement) by which the control variable is modified each time through the loop
- Definite repetition
  - Loop executes a specified number of times
  - Number of repetitions is known

# Example 3

```
int count;                      // LCV: Loop Control Variable
int num1, num2, sum;

count = 0;                      // 1. Initialize LCV
while (count < 5) {             // 2. Test LCV
    scanf("%d %d", &num1, &num2);
    sum = num1 + num2;
    printf("%d + %d = %d\n", num1, num2, sum);
    count = count + 1;          // 3. Increment LCV
}
```

**EXECUTION CHART**

| count | count<5 | repetition |
|---|---|---|
| 1 | true | 1 |
| 2 | true | 2 |
| 3 | true | 3 |
| 4 | true | 4 |
| 5 | true | 5 |
| 6 | false | |

# Loop Pitfalls

```
// Echo numbers entered back to user
printf("Enter number or zero to end: ");
scanf("%d", &num);
while (num != 0);
{
    printf("Number is %d\n\n", num);
    printf("Enter another number or zero to end: ");
    scanf("%d", &num);
}
```

```
Enter value or zero to end: 2
```

What is wrong with my program? It just sits there!

# Loop Pitfalls: Misplaced semicolon

```
// Echo numbers entered back to user
printf("Enter number or zero to end: ");
scanf("%d", &num);
while (num != 0);
{
    printf("Number is %d\n\n", num);
    printf("Enter another number or zero to end: ");
    scanf("%d", &num);
}
```

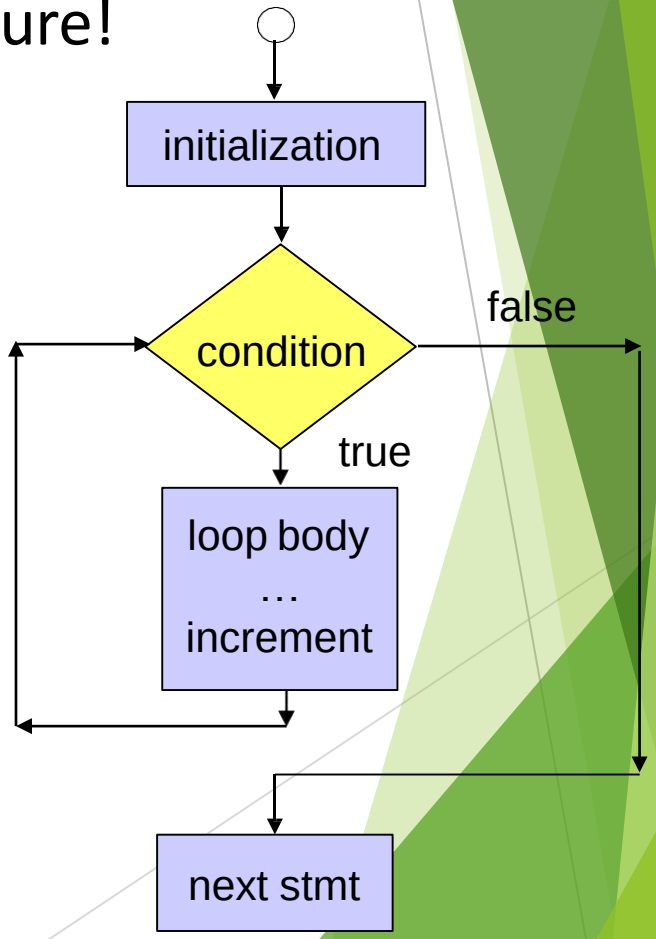
Do not place semi-colon here!

- Notice the ';' after the while condition!
  - Body of loop is between ) and ;
- Result here: **INFINITE LOOP!**

  **Ctrl-c = Kill foreground process**

# The for Repetition Structure

- A natural 'counting' loop
- Steps are built into for structure!
  1. Initialization
  2. Loop condition test
  3. Increment or decrement



# Review: Assignment Operators

- Statements of the form

  variable = variable *operator* expression;

  can be rewritten as

  variable *operator=* expression;

- Examples of assignment operators:

```
a += 5      (a = a + 5)
a -= 4      (a = a - 4)
b *= 5      (b = b * 5)
c /= 3      (c = c / 3)
d %= 9      (d = d % 9)
```

# Review: Pre-increment operator

- Pre-increment operator: ++n

i) Stand alone: add 1 to n

- If n equals 1, then after execution of the statement

```
++n;
```

- the value of n will be 2.

ii) In an expression:

- Add 1 to n and then use the new value of n in the expression.

```
printf("%d", ++n);
```

- If n is initially 1, the above statement will print the value 2.
- After execution of printf, n will have the value 2.

# Review: Post-increment operator

Pre-increment operator: n++

i) Stand alone: add 1 to n

If n equals 1, then after execution of the statement

```
n++;
```

the value of n will be 2.

ii) In an expression:

Use the value of n in the expression and then add 1 to n.

```
printf("%d", n++);
```

If n is initially 1, the above statement will print the value 1 and then add 1 to n. After execution, n will have the value 2.

# Pre- and Post-decrement operator

- Pre- and post-decrement operators, --n, n-- , behave in a similar manner
- **Use caution when using in an expression**
  - Do not use unless you know what you are doing!


# The *for* Repetition Structure

- Syntax:

```
for (initialization; test; increment)
     basic block
```

# *for* loop example

- Prints the integers from one to ten

```
int counter;
for (counter = 1; counter <= 10; counter++)
{
    printf("%d\n", counter);
}
```

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

# for Loop Example

How many times does loop body execute?

# for Loop Example

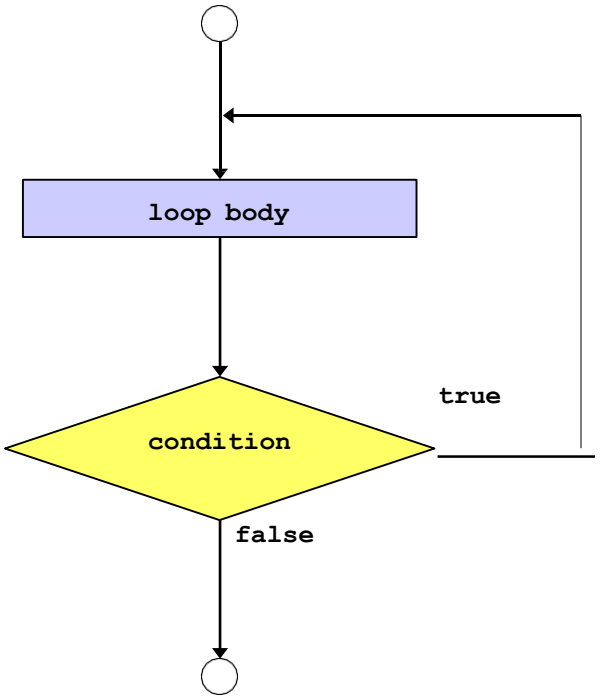
How many times does loop body execute?

```
int count;
for (count = 0; count < 3; count++) {
   printf("Bite %d  --  ", count+1);
   printf("Yum!\n");
}
```

```
Bite 1  --  Yum!
Bite 2  --  Yum!
Bite 3  --  Yum!
```

# The do-while Repetition Structure

- The **`do-while`** repetition structure is similar to the **`while`** structure
  - Condition for repetition tested after the body of the loop is executed




All actions are performed at least once!


# The do-while Repetition Structure

- Syntax:

```
do {
    statements
} while ( condition );
```

# The do-while Repetition Structure

- Example

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

- Prints the integers from **1** to **10**

# The do-while Repetition Structure

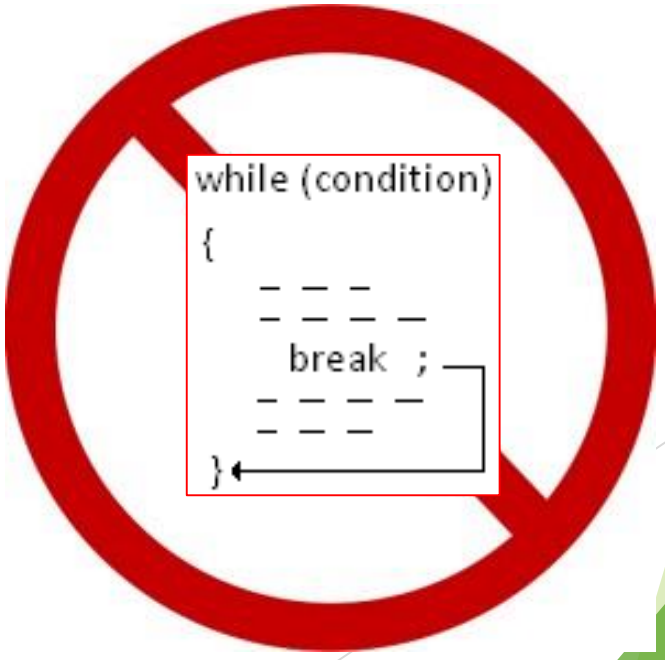
- Example

```
do {
   printf("Enter a positive weight: ");
   scanf("%d", &weight);
} while (weight <= 0);
```

  - Makes sure that the user enters a valid weight

# The break Statement

- **`break`**
  - Causes immediate exit from a **`while`**, **`for`**, **`do/while`** or **`switch`** structure
  - **We will use the break statement only to exit the switch structure!**



# Quize

- Write C program to accept number n then calculate a factorial of number n.

# Quize

- Write C program to accept number n then countdown from number n to 0 and print on screen.

# Quize

1. Write a C program to print all natural numbers from 1 to n. – using while loop

# Quize

1. Write a C program to print all natural numbers in reverse (from n to 1). – using

2. Write a C program to find sum of all natural numbers between 1 to n.

# Quize

1. Write a C program to print all even numbers between 1 to 100. – using while loop

2. Write a C program to print all odd number between 1 to 100.

3. Write a C program to find sum of all even numbers between 1 to n.

4. Write a C program to find sum of all odd numbers between 1 to n.

# Quize

1. Write a C program to print all alphabets from a to z. – using while loop

2. Write a C program to print multiplication table of any number.

3. Write a C program to count number of digits in a number.