

CONTROL STRUCTURES

CMPT 110

TENTATIVE SYLLABUS

Week	Topic
1 (Sept. 3 rd)	Introduction to Course
2 (10 th)	Introduction to Programming
3 (17 th)	Programming in VB
4 (24 th)	Events
5 (Oct. 1 st)	Representing and Storing Values
6 (8 th)	MIDTERM (Oct. 8 th)
7 (15 th)	Subprograms
8 (22 nd)	Decisions
9 (29 th)	Iteration
10 (Nov. 5 th)	Arrays
11 (12 th)	I/O
12 (19 th)	Graphics
13 (26 th)	Review

- Review Dijkstra's Control Structures.

OBJECTIVES

- Review Dijkstra's Control Structures.
- Review all control structures.

OBJECTIVES

- Review Dijkstra's Control Structures.
- Review all control structures.

OBJECTIVES

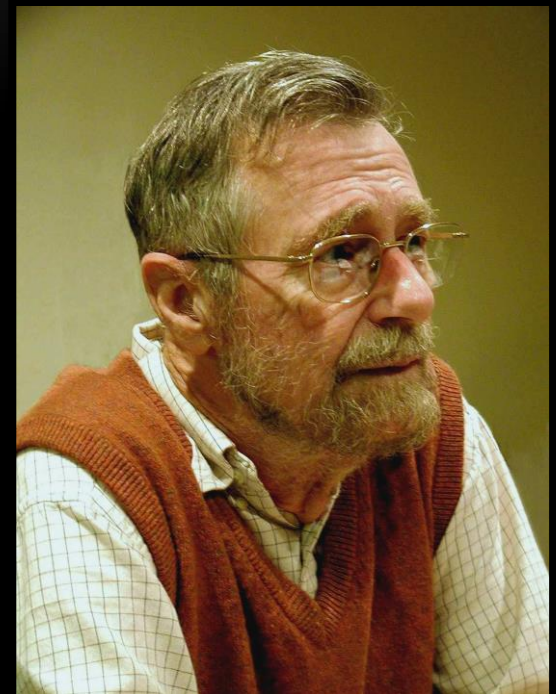
Flow of Control: The order in which individual statements, instructions or function calls are executed or evaluated.

DEFINITION

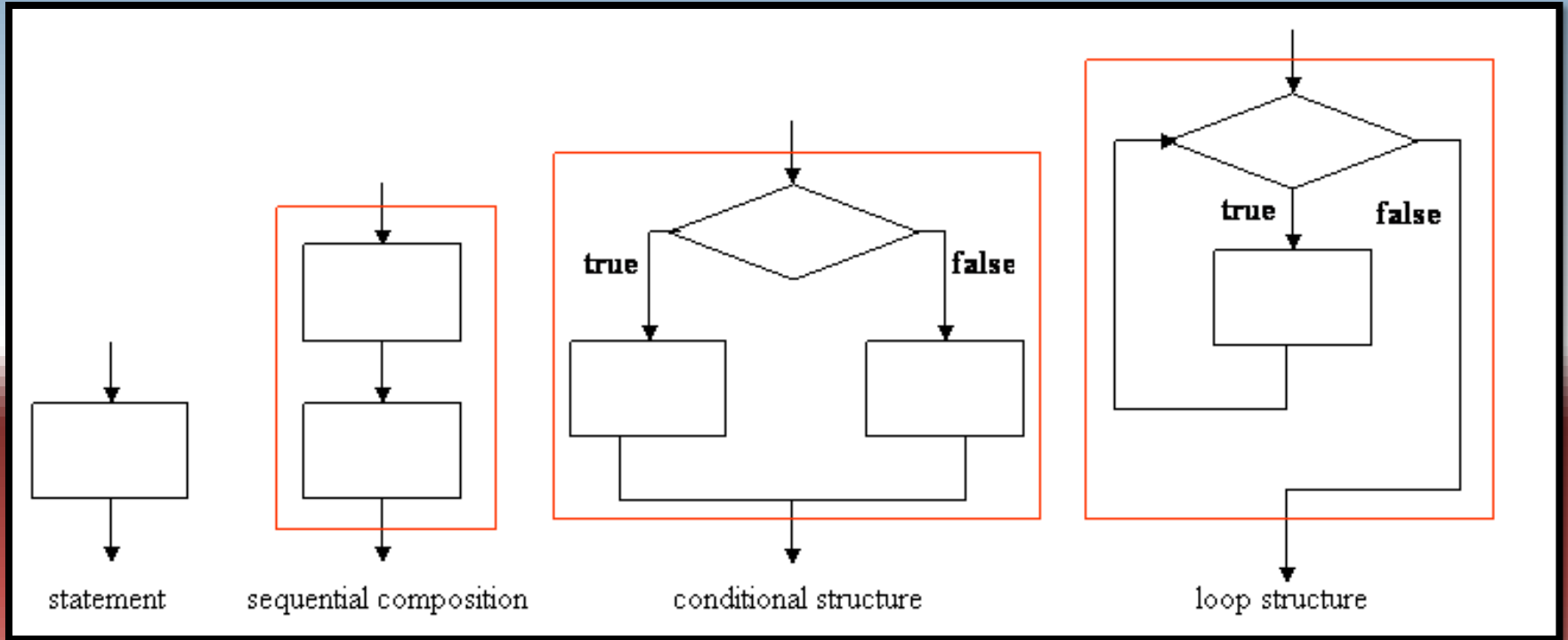
RECALL DIJKSTRA'S CONTROL STRUCTURES

Dijkstra (and others) in the late 1960's recognized that the *flow of logic* in code can be organized according to three primary ***control structures***:

1. Sequence
2. Iteration (repeating a procedure)
3. Decision



FLOW CHART DESCRIPTION



The kinds of control flow statements supported by different languages vary, but can be categorized by their effect:

CLASSIFICATION OF CONTROL STRUCTURES

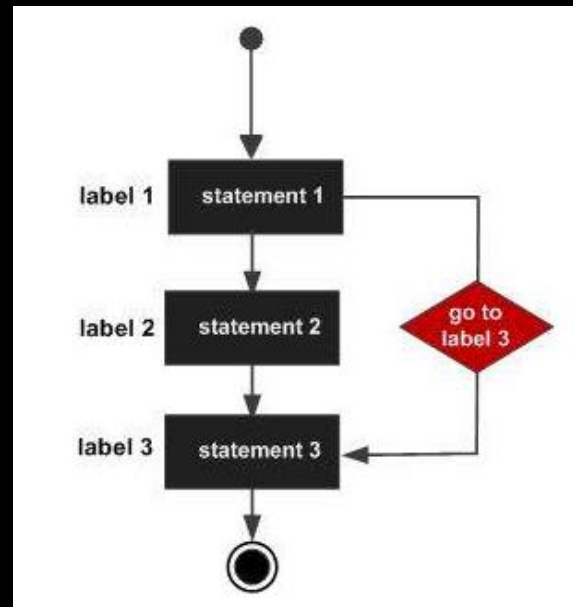
The kinds of control flow statements supported by different languages vary, but can be categorized by their effect:

- Sequential execution of statements.

CLASSIFICATION OF CONTROL STRUCTURES

The kinds of control flow statements supported by different languages vary, but can be categorized by their effect:

- Sequential execution of statements.
- Continuation at a different statement (unconditional branch or jump to some *label* – the **goto** statement),



CLASSIFICATION OF CONTROL STRUCTURES

The kinds of control flow statements supported by different languages vary, but can be categorized by their effect:

- Sequential execution of statements.
- Continuation at a different statement (unconditional branch or jump to some *label* – the **goto** statement)
- Executing a set of statements only if some condition is met (decision-
i.e., conditional branch),

CLASSIFICATION OF CONTROL STRUCTURES

The kinds of control flow statements supported by different languages vary, but can be categorized by their effect:

- Sequential execution of statements.
- Continuation at a different statement (unconditional branch or jump to some *label* – the **goto** statement,
- Executing a set of statements only if some condition is met (**decision-** i.e., conditional branch),
- Executing a set of statements zero or more times, until some condition is met (i.e., **loop** - the same as conditional branch),

CLASSIFICATION OF CONTROL STRUCTURES

The kinds of control flow statements supported by different languages vary, but can be categorized by their effect:

- Sequential execution of statements.
- Continuation at a different statement (unconditional branch or jump to some *label* – the **goto** statement),
- Executing a set of statements only if some condition is met (**decision**- i.e., conditional branch),
- Executing a set of statements zero or more times, until some condition is met (i.e., **loop** - the same as conditional branch),
- Executing a set of distant statements, after which the flow of control usually returns (subroutines, **coroutines**, and **continuations**),

CLASSIFICATION OF CONTROL STRUCTURES

The kinds of control flow statements supported by different languages vary, but can be categorized by their effect:

- Sequential execution of statements.
- Continuation at a different statement (unconditional branch or jump to some *label* – the **goto** statement),
- Executing a set of statements only if some condition is met (**decision-** i.e., conditional branch),
- Executing a set of statements zero or more times, until some condition is met (i.e., **loop** - the same as conditional branch),
- Executing a set of distant statements, after which the flow of control usually returns (**subroutines, coroutines, and continuations**),
- Stopping the program, preventing any further execution (**unconditional halt**).

CLASSIFICATION OF CONTROL STRUCTURES

Structured program theorem

Structured program theorem

From Wikipedia, the free encyclopedia

The **structured program theorem**, also called **Böhm-Jacopini theorem**,^{[1][2]} is a result in [programming language theory](#). It states that a class of [control flow graphs](#) (historically called [charts](#) in this context) can compute any [computable function](#) if it combines subprograms in only three specific ways ([control structures](#)). These are

1. Executing one subprogram, and then another subprogram (sequence)
2. Executing one of two subprograms according to the value of a [boolean expression](#) (selection)
3. Executing a subprogram as long as a boolean expression is true (iteration)

ASIDE: PROOF OF STRUCTURED PROGRAMMING THEOREM

Say you have a program that consists of a sequence of statements S_i ; prefix each statement with a label $S'_i \leftarrow L_i : S_i$ and update existing goto statements to point to the right locations. Declare a location variable, $l \leftarrow 1$, and wrap the prefixed statements in a while loop that will continue until the last statement is reached, **while**($l \neq M$) **do** S' .

Apply the following rewrite rules to S' :

- Goto rule: Replace L_i **goto** L_j with **if** ($l = i$) **then** $l \leftarrow j$
- If-goto rule: Replace $L_i : \text{if (cond) then goto } L_j$ with **if** ($l = i \wedge (\text{cond})$) **then** $l \leftarrow j$ **else** $l \leftarrow l + 1$
- Otherwise rule: Replace $L_i : S_i$ with **if** $l = i$ **then** S_i , $l \leftarrow l + 1$

The resulting program is then free of goto statements showing the correspondence between the two paradigms.

FYI – CS Students

See: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.119.9119&rep=rep1&type=pdf>

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: that a process, considered as a time succession

dynamic progress is only characterized when we call of the procedure we refer. With the line we can characterize the progress of the process by textual indices, the length of this sequence is the dynamic depth of procedure calling.

Let us now consider repetition clauses (like *do* or *repeat A until B*). Logically speaking, such clauses are superfluous, because we can express repetition by recursive procedures. For reasons of realism we shall include them: on the one hand, repetition clauses are mentioned quite comfortably with present day programming; on the other hand, the reasoning pattern known as *loop invariants* makes us well equipped to retain our intellectual processes generated by repetition clauses. Yet, the repetition clauses textual indices are not sufficient to describe the dynamic progress of the process. A repetition clause, however, we can associate with a "dynamic index," inexorably counting the occurrences of the corresponding current repetition. As repetition clauses (or procedure calls) may be applied nestingly, the progress of the process can always be unique (mixed) sequence of textual and/or dynamic indices.

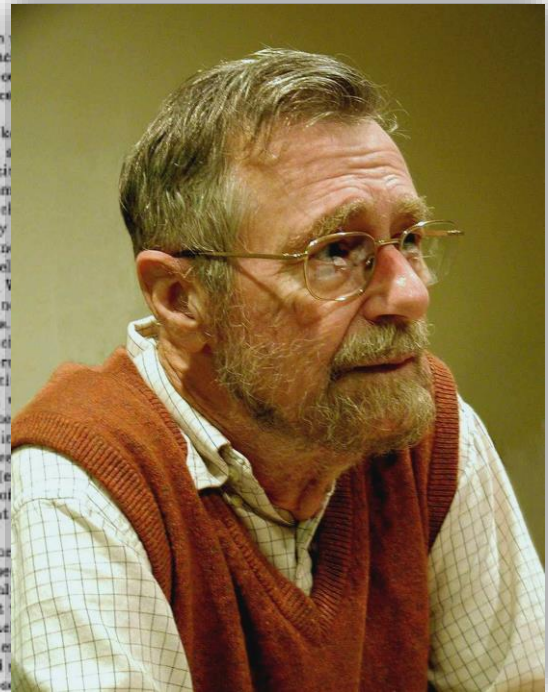
The main point is that the values of these indices are under the programmer's control; they are generated (either by the programmer or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates to describe the progress of the process.

Why do we need such independent coordinates? The answer is—and this seems to be inherent to sequencing—because we can interpret the value of a variable only in the context of the progress of the process. If we wish to count the number of people in an initially empty room, we can assign a value of one whenever we see someone enter the room between moments that we have observed someone leave the room but have not yet performed the subtraction. Its value equals the number of people in the room minus one!

The unbridled use of the go to statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the go to statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (via a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, n equals the number of persons in the room minus one!

The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as befitting its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. As I to



OPERATORS IN VISUAL BASIC

OPERATORS IN VISUAL BASIC

Operators	
Arithmetic	^, -, *, /, \, Mod, +
Assignment	=, ^=, *=, /=, \=, +=, -=, &=
Comparison/Relational	=, <>, <, >, <=, >=, Like, Is
Concatenation	&, +
Logical/bitwise	Not, And, Or, Xor, AndAlso, OrElse
Miscellaneous operations	AddressOf, GetType

DECISION STRUCTURE



OPERATORS IN VISUAL BASIC

Operators	
Arithmetic	\wedge , -, *, /, \, Mod, +
Assignment	=, \wedge =, * =, / =, \ =, + =, - =, & =
Comparison/ Relational	=, <>, <, >, <=, >=, Like, Is
Concatenation	&, +
Logical/bitwise	Not, And, Or, Xor, AndAlso, OrElse
Miscellaneous operations	AddressOf, GetType

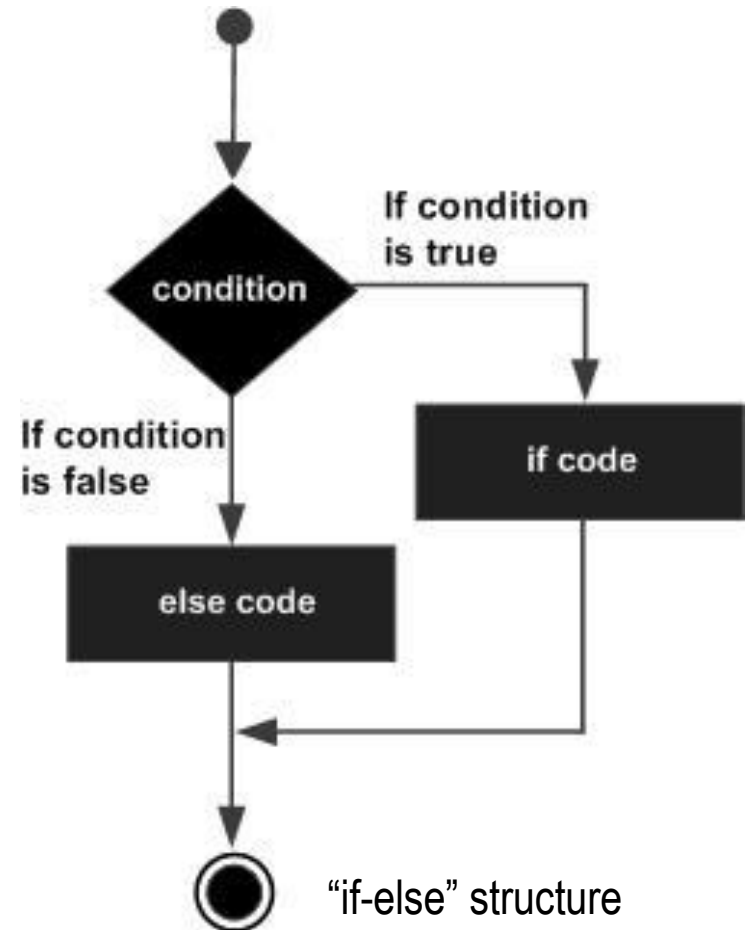
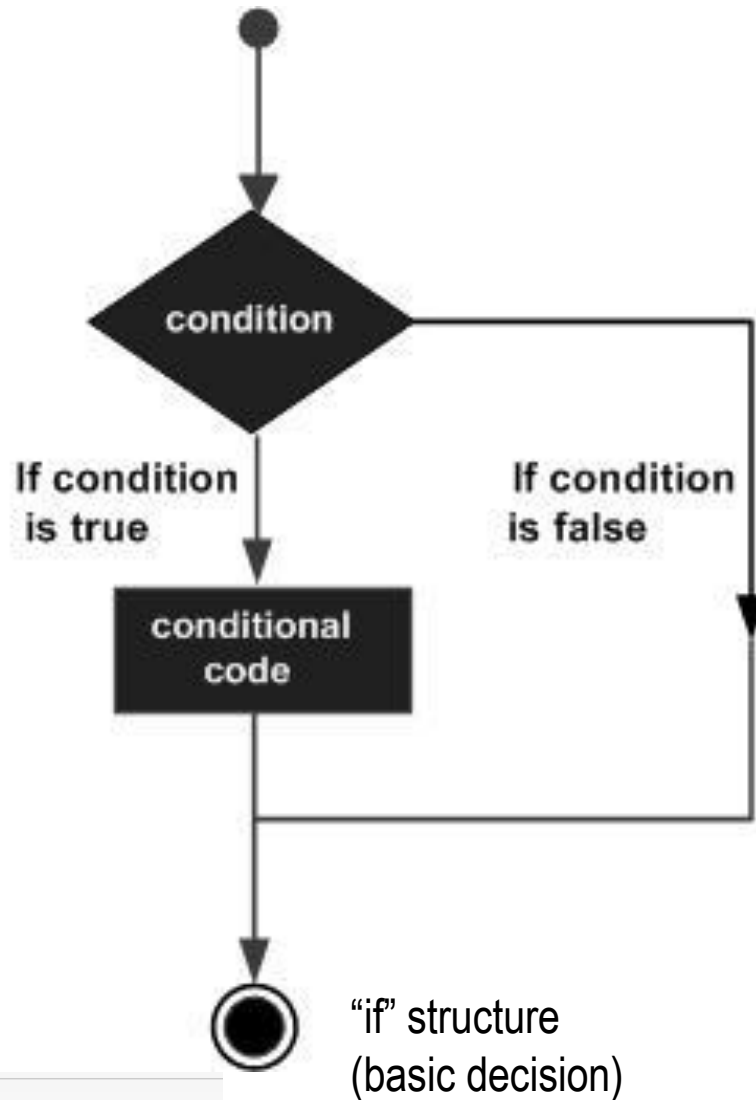
DECISIONS ARE PART OF LIFE

Wine Flowchart



grimm®

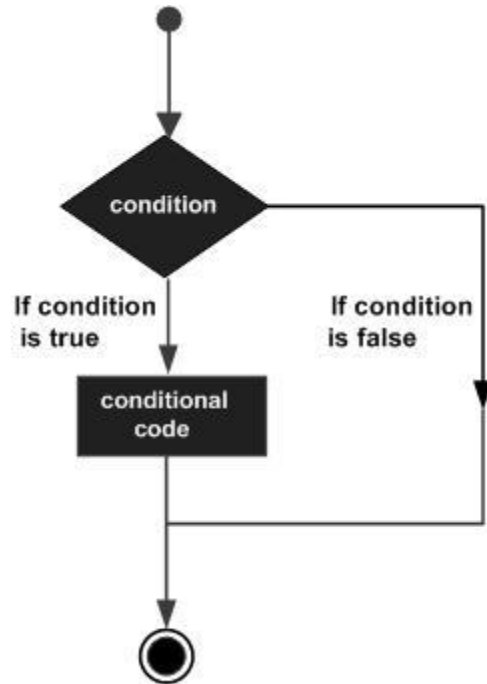
DECISION STRUCTURE FLOW CHARTS



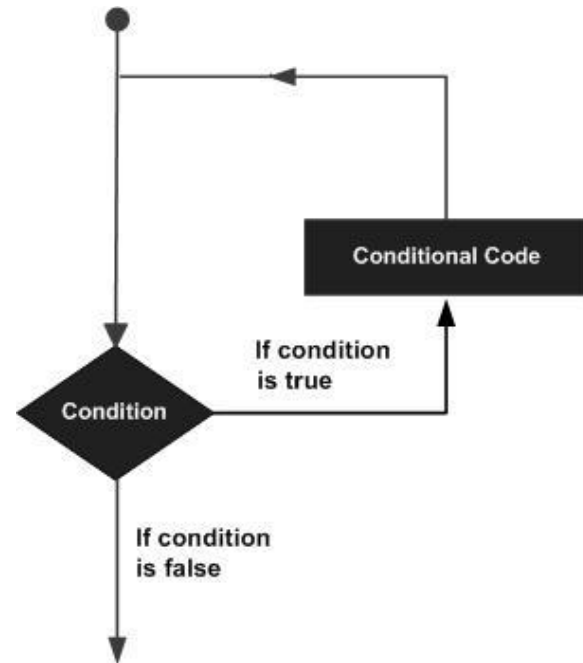
ITERATION (LOOP) STRUCTURE



LOOP STRUCTURE FLOW CHART







If Decision Structure



While Loop Structure (multiple ifs)

DIFFERENT CLASSES OF LOOP STRUCTURES

Sr.No	Loop Type & Description
1	while loop  Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	for loop  Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	do...while loop  Like a 'while' statement, except that it tests the condition at the end of the loop body.
4	nested loops  You can use one or more loop inside any another 'while', 'for' or 'do..while' loop.

INFINITE LOOP IN C++

The Infinite Loop

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the 'for' loop are required, you can make an endless loop by leaving the conditional expression empty.

```
#include <iostream>
using namespace std;

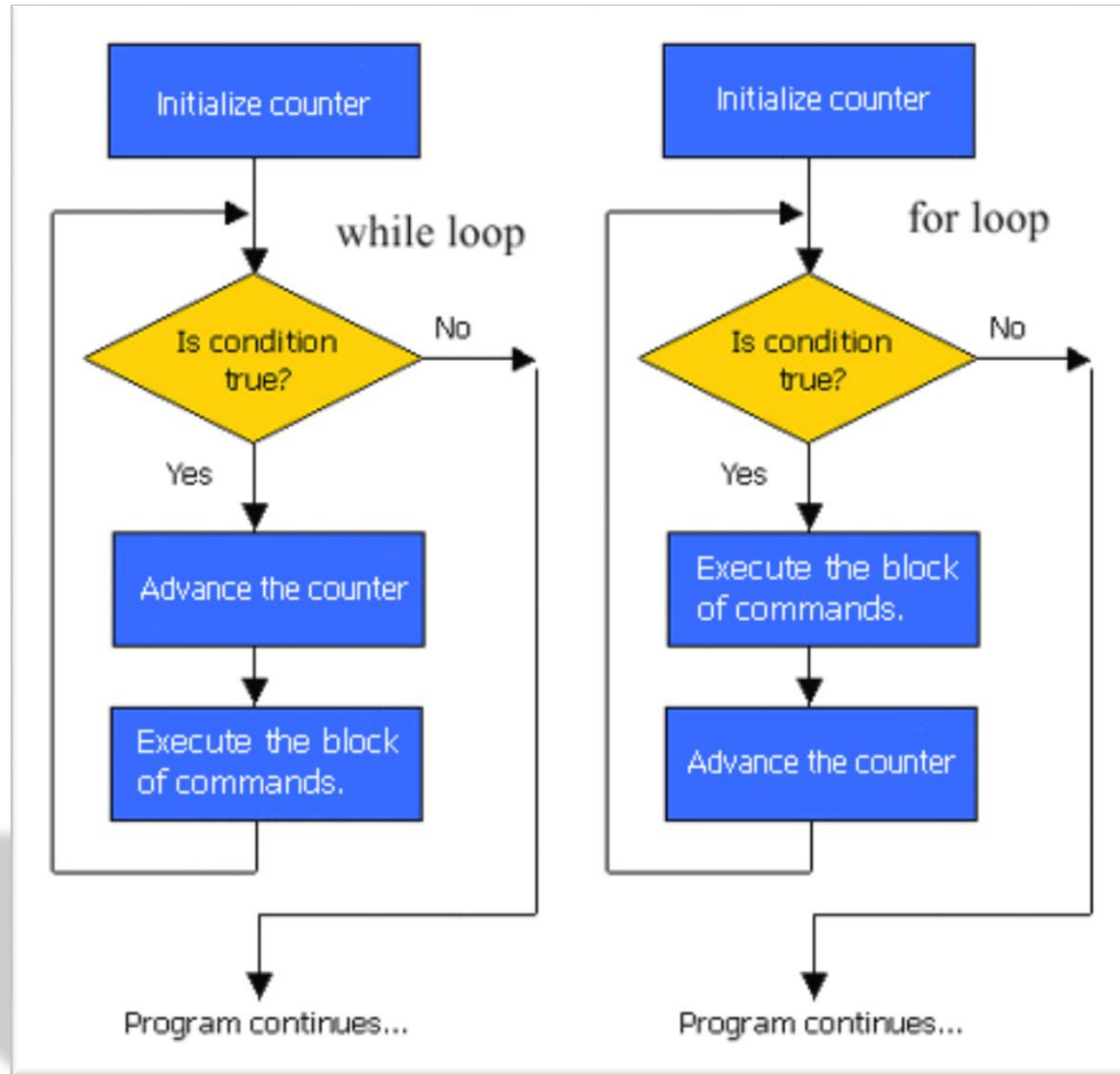
int main () {
    for( ; ; ) {
        printf("This loop will run forever.\n");
    }

    return 0;
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C++ programmers more commonly use the 'for (;;)' construct to signify an infinite loop.

NOTE – You can terminate an infinite loop by pressing Ctrl + C keys.

WHILE-LOOPS VERSUS FOR-LOOPS



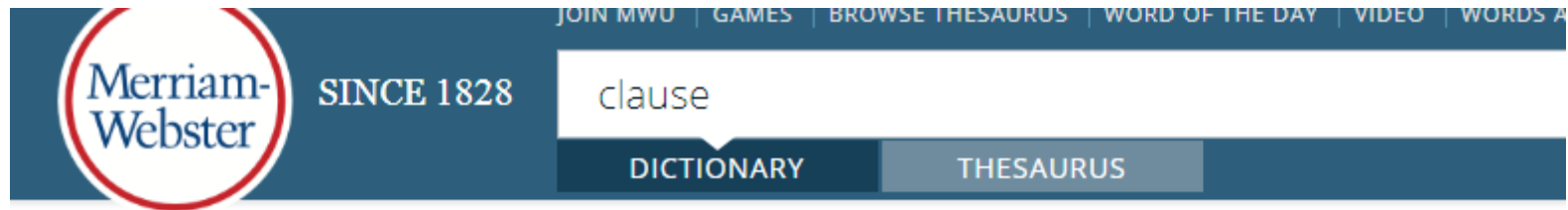
WHILE-LOOP VERSUS FOR-LOOP

- **For-loops** are used when you know exactly how many iterations are involved.

WHILE-LOOP VERSUS FOR-LOOP

- **For-loops** are used when you know exactly how many iterations are involved.
- **While-loops** are more flexible (iterate until a condition is met, for example).
 - These are also classified as *iterative loops* (for-loop) and *conditional loops* (while-loop).

NOTE: CLAUSE



clause noun

\ 'klôz  \

Definition of *Clause*

- 1** : a group of words containing a subject and predicate and functioning as a member of a complex (see [COMPLEX entry 2 sense 1b\(2\)](#)) or compound (see [COMPOUND entry 2 sense 3b](#)) sentence

// The sentence "When it rained they went inside" consists of two *clauses*: "when it rained" and "they went inside."

- 2** : a separate section of a discourse (see [DISCOURSE entry 1 sense 2](#)) or writing
specifically : a distinct article in a formal document

// a *clause* in a contract

PRESENTATION TERMINATED

