Programming Language

Statement-Level Control Structures

```
#include <stdio.h>
//chapter 9
double addDouble(double a, double b){
    try{ // chapter 14 (try-catch)
        a - a/b;
    }catch(...){
        cout << "divide by zero";
    }
    return a | b;
}
int main(){
    // chapter 5,6
    int x = 5;
    double y;

    // chapter 7
    x = y * 6 + 1 * x;

    // chapter 8
    if(x<10){
        // chapter 9
        y = addDouble(2.5, 1.2);
    }
    // chapter 9
    return 0;
}
```

Chapter 5. Names, Bindings, and Scopes

Chapter 6. Data Types

Chapter 7. Expressions and Assignements Statements

Chapter 8. Statement-Level Control Structures

Chapter 9. Subprograms

Contents

Introduction (1.)

Structured programming

- A programming paradigm that improves computer programs in terms of:
  - Clarity
  - Quality
  - Development time
- A code block is structured when it has a single entry point and single exit point, as shown in the flowchart or NS diagrams.
- Structured programming is a method that makes it easier to verify program correctness.

Control structure

- A control statement and the collection of statements whose execution it controls

Example:

```
if (x > 0) {          // control statement
    y = x * 2;        // statements being controlled
    z = y + 1;
}
```

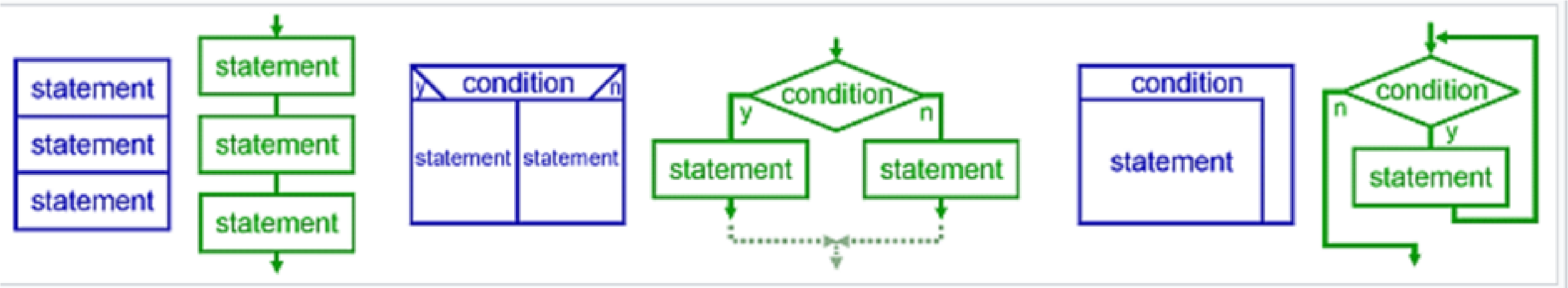
- Characteristics in structured programming:
  - Single entry point
  - Single exit point

Three types of control structures:

- Sequence: statements execute one after another
- Selection (if-then-else): choose between alternative control flow paths
- Iteration (loops): repeated execution of statements

Visual representation:

- Nassi–Shneiderman diagram (blue) https://en.wikipedia.org/wiki/Nassi–Shneiderman_diagram
- Flow charts (green)



Sequence If-then-else Repetition

Computations

- Imperative language
  - Computations are accomplished by evaluating expressions and assigning resulting values to variables
  - To make programs flexible and powerful, two additional mechanisms are needed:
    - Selection: choosing among alternative control flow paths
    - Iteration: repeated execution of statements or sequences of statements
  - These are called **control statements**
- Functional languages:
  - Computations are accomplished by evaluating expressions and applying functions to given parameters
  - The flow of execution is controlled by other expressions and functions

Imperative languages (C, Java, Python):

```c
// Computation by assignment and control flow
int sum = 0;
for (int i = 1; i <= 10; i++) {
    sum = sum + i;  // assigning values
}
// Result: sum = 55
```

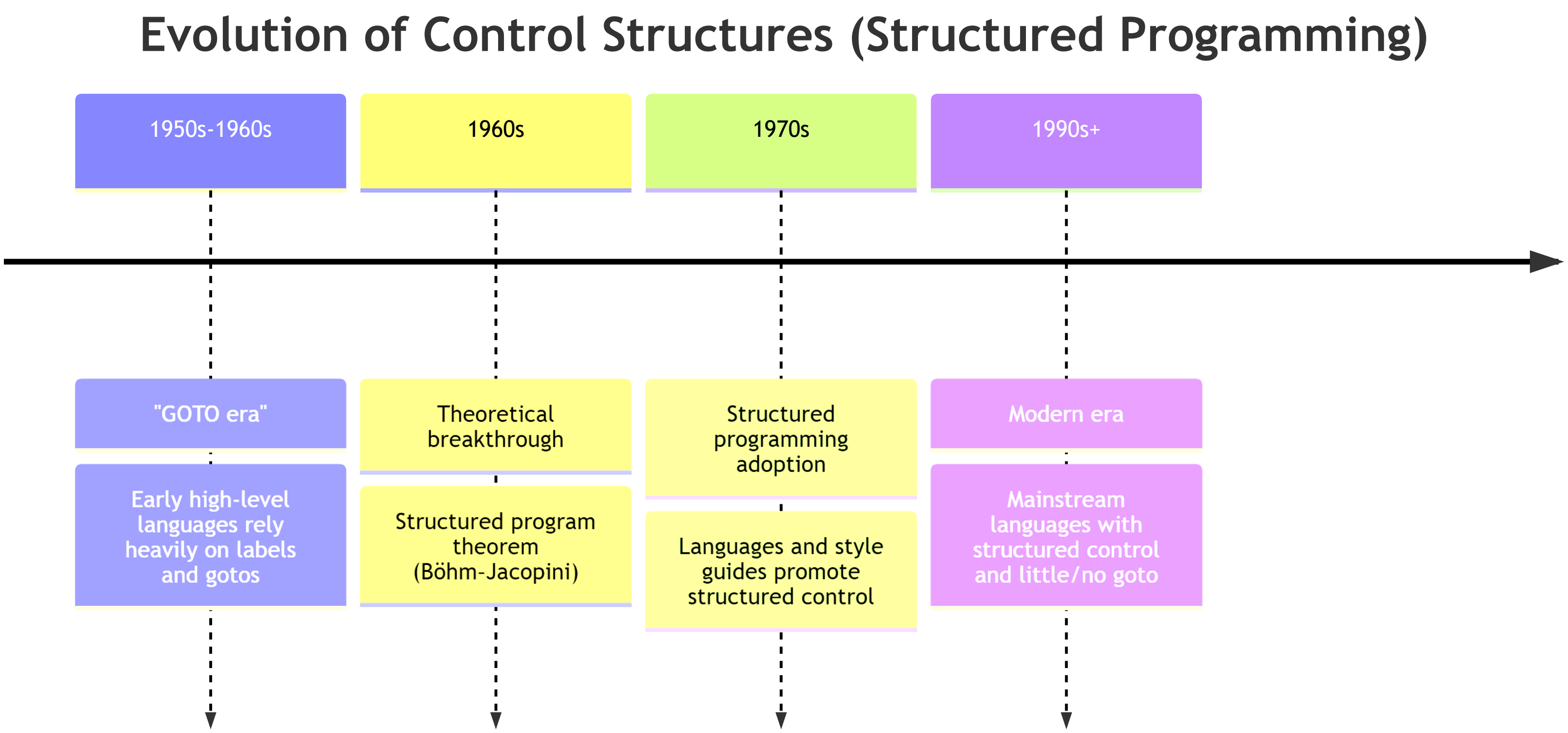
Functional languages (Haskell, Lisp):

```haskell
-- Computation by applying functions
sum [1..10]  -- applying the sum function
-- Result: 55
```

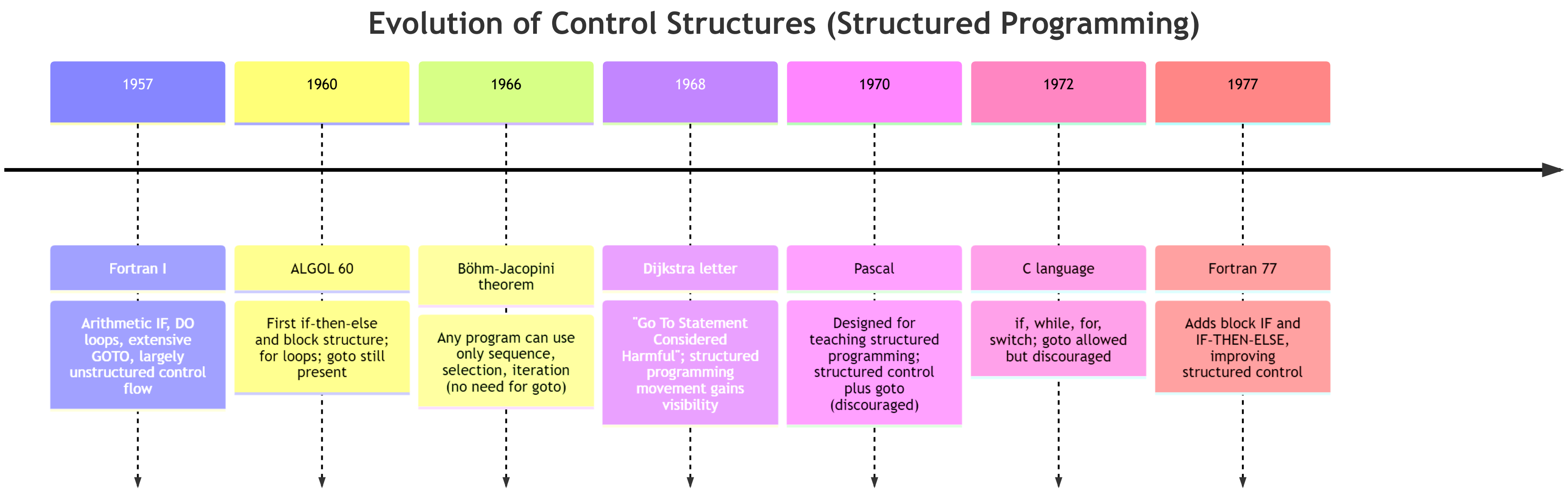
Both accomplish the same computation (adding numbers 1-10), but:

- Imperative: uses assignments and control statements (for loop)
- Functional: uses function application (sum function)

Historical progression:



Historical progression:



Historical development

- Fortran was the first successful programming language, designed for the IBM 704
- Its control statements were directly related to machine language instructions
- At the time, little was known about the difficulty of programming
- By today's standards, Fortran's control statements are seriously lacking

"Little was known about the difficulty"

Explain

- In 1957, programmers thought this Fortran code was fine!

Why?

- Coming from assembly: everything was jumps/branches
- No large programs yet to see the mess
- No experience with maintenance

Fortran Example

https://en.wikibooks.org/wiki/Fortran/Fortran_examples

```
C AREA OF A TRIANGLE - HERON'S FORMULA
C INPUT - CARD READER UNIT 5, INTEGER INPUT
C OUTPUT -
C INTEGER VARIABLES START WITH I,J,K,L,M OR N
      READ(5,501) IA,IB,IC
  501 FORMAT(3I5)
      IF (IA) 701, 777, 701
  701 IF (IB) 702, 777, 702
  702 IF (IC) 703, 777, 703
  777 STOP 1
  703 S = (IA + IB + IC) / 2.0
      AREA = SQRT( S * (S - IA) * (S - IB) * (S - IC) )
      WRITE(6,801) IA,IB,IC,AREA
  801 FORMAT(4H A= ,I5,5H  B= ,I5,5H  C= ,I5,8H  AREA= ,F10.
     $13H SQUARE UNITS)
      STOP
      END
```

```
IF (IA) 701, 777, 703
```

Meaning:

- If IA < 0 → goto line 701
- If IA = 0 → goto line 777 (STOP - error!)
- If IA > 0 → goto line 703

Theoretical research (1960s-1970s):

- Böhm and Jacopini (1966) proved that although a simple goto statement is minimally sufficient, a well-designed language needs only two types of control statements:
  - One for choosing between two control flow paths (selection)
  - One for logically controlled iterations (loops)
- This means the unconditional branch (`goto`) is superfluous—potentially useful but not essential

- Only `goto` is hard to read

https://onlinegdb.com/Y-QG2OCki

```c
#include <stdio.h>
int main(){
    int i=0, j=5;
    printf("=== With goto (hard to read) ===\n");
    i = 0;
    loop:
        printf("i = %d\n", i);
        i++;
        if(i < j) goto loop;

    printf("\n=== With while (clear!) ===\n");
    i = 0;
    while(i < j) {
        printf("i = %d\n", i);
        i++;
    }
    printf("\n*** Both produce the same output! ***\n");
    printf("*** goto is SUPERFLUOUS (not needed)! ***\n");
    return 0;
}
```

Writability vs. Readability trade-off:

- More control statements → Programs are easier to write
  - Example: Using a for loop for counting is easier than using a while loop
- Too many control statements → Programs become harder to read
  - Programmers must learn more statements
  - Readers may not know all the statements used

Selection Statements (2.)

A selection statement provides the means of choosing between two or more execution paths in a program.

- One-way selection
- Two-way selection
- N-way selection or multiple selection

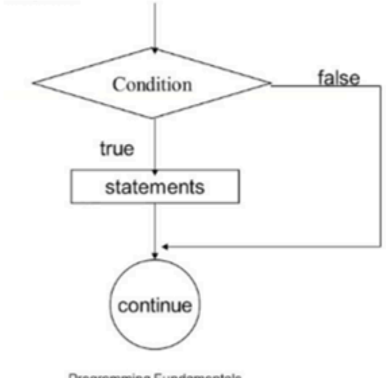
$$
\begin{aligned}
\langle\text{if\_stmt}\rangle \rightarrow\ & \mathbf{if}\ \langle\text{logic\_expr}\rangle\ \mathbf{then}\ \langle\text{stmt}\rangle \\
\mid\ & \mathbf{if}\ \langle\text{logic\_expr}\rangle\ \mathbf{then}\ \langle\text{stmt}\rangle\ \mathbf{else}\ \langle\text{stmt}\rangle
\end{aligned}
$$

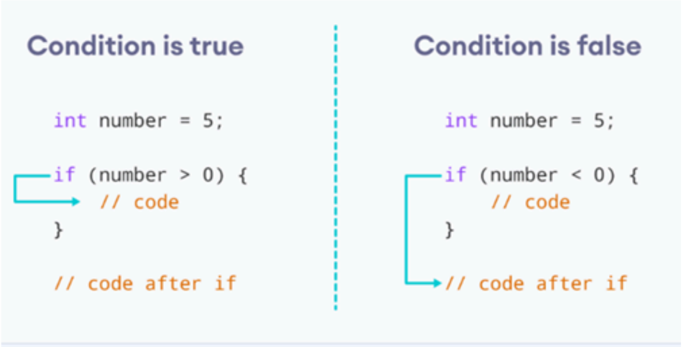
One-way selection

- General form:

```
if control_expression
    then clause
```

- Control expression:
  - In C89, C99, Python, and C++, the control expression can be arithmetic
  - In languages such as Ada, Java, Ruby, and C#, the control expression must be Boolean





Then clause

- In contemporary languages, then clauses can be single or compound statements

- Ruby

```
if conditional [then]
   code...
end
```

https://onlinegdb.com/MXEpt_747

```
score = 85

if score >= 80 then puts "Grade B" end # then is required

if score >= 80
    puts "Grade B"
end
```

Then clause

- In Perl, all clauses must be delimited by braces (they must be compound even if there is only 1 statement)

```
my $a = 1;
if($a == 1) {
   print("Welcome to the Perl if tutorial!\n");
   print("another form of the Perl if statement\n");
}
```

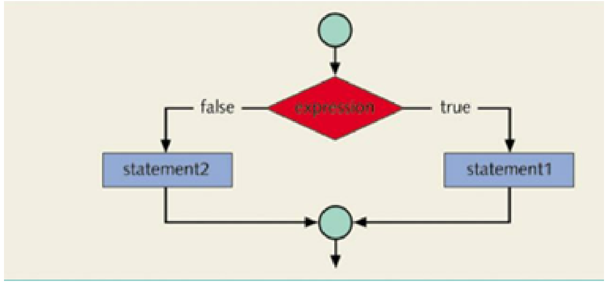
- Python uses **indentation** to define clauses

```
if x > y :
        x = y
        print "case 1"
```

Two-Way selection

```
if control_expression
  then clause
  else clause
```

```
cin >> option ;  cost = 10;
 if  ( option==1 ) {
          cost += 10 ;
 } else {
       cost += 20 ;
 }
```




```
cin >> option ;  cost = 10;
 if  ( option==1 )
          cost += 10 ;
  else
          cost += 20 ;
```

Dangling else

```
if(sum==0)
    result = 0;
else
    result = 1;
```

```
if (sum == 0) // (1)
  if (count == 0) // (2)
    result = 0;
else // which if clause (1) or (2) are match?
  result = 1;
```

- Which `if` gets the `else`?
- The Java, C, C++ and C# can ignore braces `{}` that might be ambiguous **(dangling else)**
- Java's static semantics rule: `else` matches with the nearest `if` (including C, C++, and C#)

Nesting Selectors

- To force an alternative semantics, compound statements may be used:

```
if (sum == 0) {
    if (count == 0)
        result = 0;
}
else result = 1;
```

- The above solution is used in C, C++, and C#
- Perl requires that all then and else clauses to be compound and avoid the above problem

For Perl

- But Perl is always require braces `{}` to avoid ambigous **(dangling else)**

Case 1

```
if (sum == 0){
    if (count == 0){
        result = 0;
    }
} else {
    result = 1;
}
```

Case 2

```
if (sum == 0){
    if (count == 0){
        result = 0;
    }
    else {
        result = 1;
    }
}
```

For Ruby

- The problem can also be solved by alternative means of forming compound statements, e.g., using a special word `end` in Ruby

Case 1

```ruby
if sum == 0 then
  if count == 0 then
    result = 0
  else
    result = 1
  end
end
```

Case 2

```ruby
if sum == 0 then
  if count == 0 then
    result = 0
  end
else
  result = 1
end
```

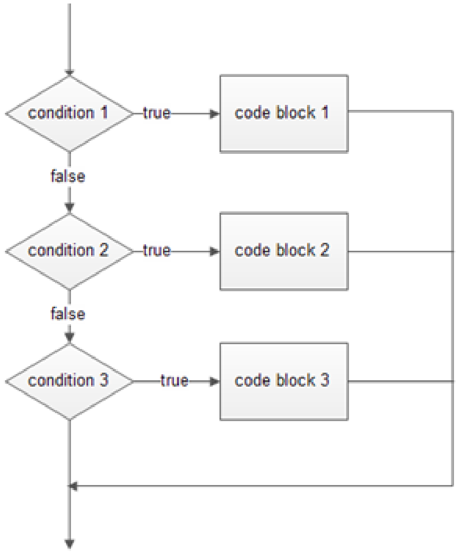
Multiple-Way Selection Using if

- Multiple selectors can appear as direct extensions to two-way selectors, using else-if clauses, for example in Python:

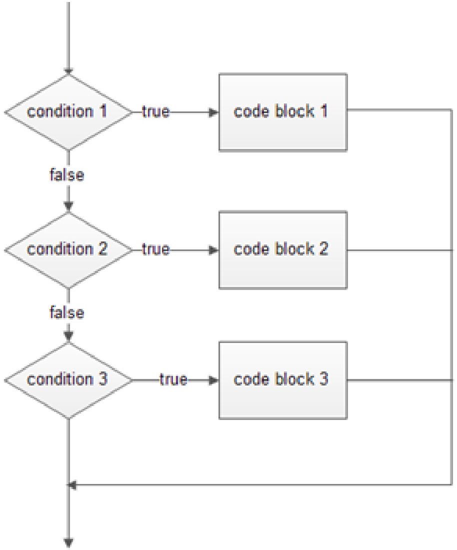
```
if count < 10 :
    bag1 = True
elif count < 100 :
    bag2 = True
elif count < 1000 :
    bag3 = True
```

- More readable than deeply nested two-way selectors!

- Can compare ranges



Add else to prevent dangling else




```
cin >> option ;  cost = 10;
 if  ( option==1 ) {
        cost += 10 ;
} else if ( option==2 ) {
                cost += 20 ;
          } else if ( option==3 ) {
                cost += 30 ;
          }
          else {
                cost += 40 ;
          }
```

Ignore braces

```
cin >> option; cost = 10;
if (option==1) {
    cost += 10;
} else if (option==2) {
    cost += 20;
} else if (option==3) {
    cost += 30;
} else {
    cost += 40;
}
```

```
cin >> option ;  cost = 10;
if  ( option==1 )
    cost += 10 ;
else if ( option==2 )
    cost += 20 ;
else if ( option==3 )
    cost += 30 ;
else
    cost += 40 ;
```

Implement with Switch?

```cpp
#include <iostream>
using namespace std;
int main () {
   // local variable declaration:
   int a;
   cin >> a;
   // check the boolean condition
   if( a == 10 ) {
      // if condition is true then print the following
      cout << "Value of a is 10" << endl;
   } else if( a == 20 ) {
      // if else if condition is true
      cout << "Value of a is 20" << endl;
   } else if( a == 30 ) {
      // if else if condition is true
      cout << "Value of a is 30" << endl;
   } else {
      // if none of the conditions is true
      cout << "Value of a is not matching" << endl;
   }
   cout << "Exact value of a is : " << a << endl;

   return 0;
}
```

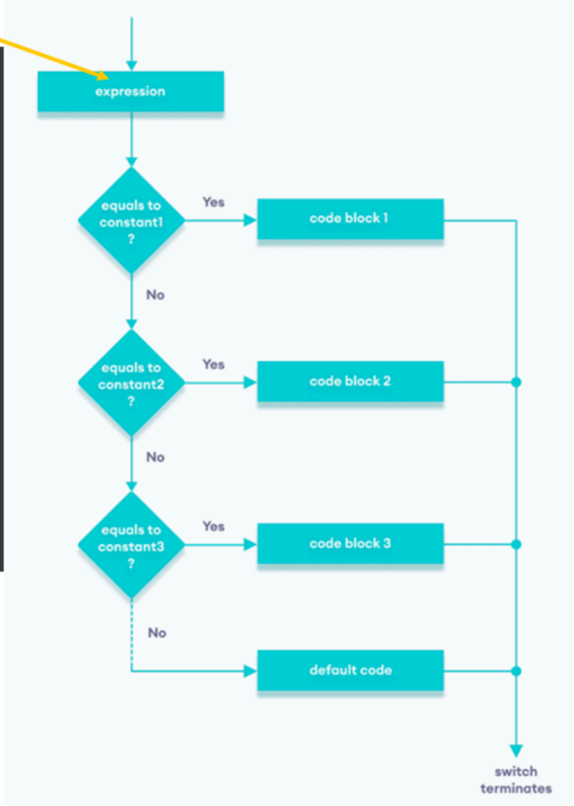
Switch in C, C++, Java

- Design choices for C's `switch` statement
  - Control expression can be only an integer type
  - Selectable segments can be statement sequences, blocks, or compound statements
  - Any number of segments can be executed in one execution of the construct (there is no implicit branch at the end of selectable segments); `break` is used for exiting `switch` → reliability of missing `break`
  - `default` clause is for unrepresented values (if there is no `default`, the whole statement does nothing)

Switch in C, C++, Java

variable or an integer expression

```
switch (expression)  {
    case constant1:
        // code to be executed if
        // expression is equal to constant1;
        break;

    case constant2:
        // code to be executed if
        // expression is equal to constant2;
        break;
        .
        .
        .
    default:
        // code to be executed if
        // expression doesn't match any constant
}
```



Switch vs. If-ElseIf-Else

```
using namespace std;

int main () {
     int a;
    cin >> a;
    switch(a)
    {
      case  10 :
                cout << "Value of a is 10" << endl;
                break;
      case 20 :
              cout << "Value of a is 20" << endl;
              break;
      case 30 :
              cout << "Value of a is 30" << endl;
              break;
      default:
              cout << "Value of a is not matching" << endl;
    }
    cout << "Exact value of a is : " << a << endl;

    return 0;
}
```

```
#include <iostream>
using namespace std;
int main () {

   int a;
   cin >> a;
   if( a == 10 ) {
      cout << "Value of a is 10" << endl;
   } else if( a == 20 ) {
      cout << "Value of a is 20" << endl;
   } else if( a == 30 ) {
      cout << "Value of a is 30" << endl;
   } else {
      cout << "Value of a is not matching" << endl;
   }
   cout << "Exact value of a is : " << a << endl;

   return 0;
}
```

Without `break`

```cpp
#include <iostream>

using namespace std;

int main () {
     int a;
   cin >> a;
   switch(a)
   {
     case  10 :
              cout << "Value of a is 10" << endl;
     case 20 :
            cout << "Value of a is 20" << endl;
            break;
     case 30 :
            cout << "Value of a is 30" << endl;
            break;
     default:
            cout << "Value of a is not matching" << endl;
   }
   cout << "Exact value of a is : " << a << endl;
   return 0;
}
```

```cpp
#include <iostream>
using namespace std;
int main () {

   int a;
   cin >> a;
   if( a == 10 ) {
      cout << "Value of a is 10" << endl;
      cout << "Value of a is 20" << endl;
   } else if( a == 20 ) {
      cout << "Value of a is 20" << endl;
   } else if( a == 30 ) {
      cout << "Value of a is 30" << endl;
   } else {
      cout << "Value of a is not matching" << endl;
   }
   cout << "Exact value of a is : " << a << endl;

   return 0;
}
```

Duff’s device

- This code uses a very unusual and confusing control flow pattern called "Duff’s device" - mixing switch cases with if-else statements. It’s legal C++ but very bad practice!

```cpp
#include <iostream>
using namespace std;
 void process_prime(int x)
 { cout << "process_prime(x) -> " <<  x << endl;}
 void process_composite(int x)
 { cout << "process_composite(x) -> " <<  x << endl; }
 bool prime(int x) {
     if( x >= 0 && x <= 7) {
         cout << "prime(x) -> true" <<  x << endl;
        return true ;
     }
    else {
        cout << "prime(x) -> false" <<  x << endl;
        return false;
     }
 }
```

```cpp
int main () {
    int x=1;
    while(x!=-1)
    {
        cout << "input x = " ;
        cin >> x;
        switch (x)
          default:
            if (prime(x))
               case 2: case 3: case 5: case 7:
                process_prime(x);
            else
              case 4: case 6: case 8:
              case 9: case 10:
                process_composite(x);
    }
   return 0;
}
```

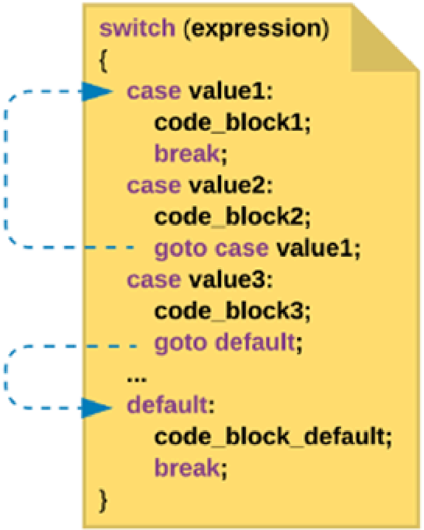
https://www.geeksforgeeks.org/c/duffs-device-work/

For Java

```
public class MyClass {


    public static void main(String args[]) {
      int x=2,z=2;
        switch (x)
        {
            case 1:
                System.out.println("Sum of x+y = " + z);
                break;
            case 2:
                System.out.println("Sum of x+y1 = " + z);
              //  break;
            case 3:
                System.out.println("Sum of x+y2 = " + z);
                break;

        }

      System.out.println("Sum of x+y = " + z);
    }
}
```

Multiple-Way Selection in C#

- It has a static semantics rule that disallows the implicit execution of more than one segment
  - Each selectable segment must end with an unconditional branch (goto or break)



goto in switch

goto with label

Multiple-Way Selection in C#

```
using System;
public class MainClass {
  public static void Main() {
    for(int i=1; i <= 5; i++) {
      switch(i) {
        case 1:
          Console.WriteLine("In case 1 , i={0}", i );
          //goto case 3; // uncomment
        case 2:
          Console.WriteLine("In case 2, i={0}", i);
          goto case 1;
        case 3:
          Console.WriteLine("In case 3, i={0}", i);
          goto default; // try comment
        case 4:
          Console.WriteLine("In case 4, i={0}", i);
          break;
        default:
          Console.WriteLine("In default, i={0}", i);
          break;
      }
      Console.WriteLine();
    }
  }
}
```

Multiple-Way Selection in Swift

- More reliable than C's switch
  - Once a stmt_sequence execution is completed, control is passed to the first statement after the case statement

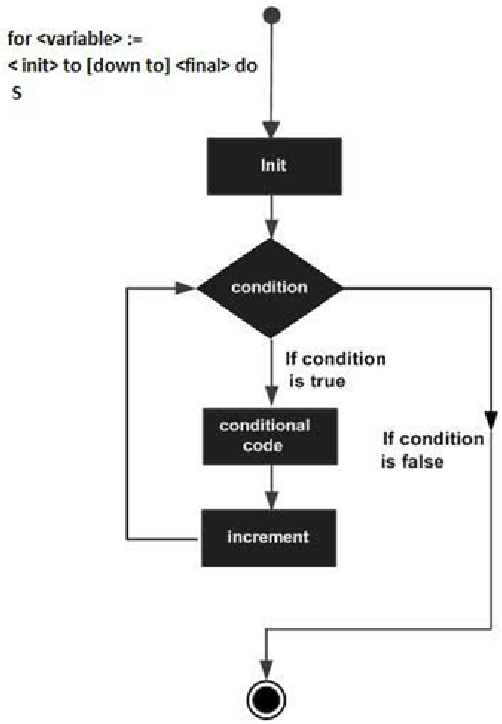
```
let marks = 80

switch marks {
  case 90...100:
    print("Wonderful")

  case 70..<90:
    print("Very Good")

    case 35..<70:
    print("Scope of improvement")

  case ..<35:
    print("Need immediate attention !")

  default:
    print("Invalid Marks")
}
```

Iterative Statements (3.)

- The repeated execution of a statement or compound statement is accomplished either by iteration or recursion

The designer must answer these:

- How is the iteration controlled?
  - Use logical, counting or both.
- Where should the control mechanism appear in the loop statement?
  - It can be anywhere but, the mechanism is executed and affects before or after execution of the statement’s body.



Type of Loops

1. Counter-Controlled (know count)

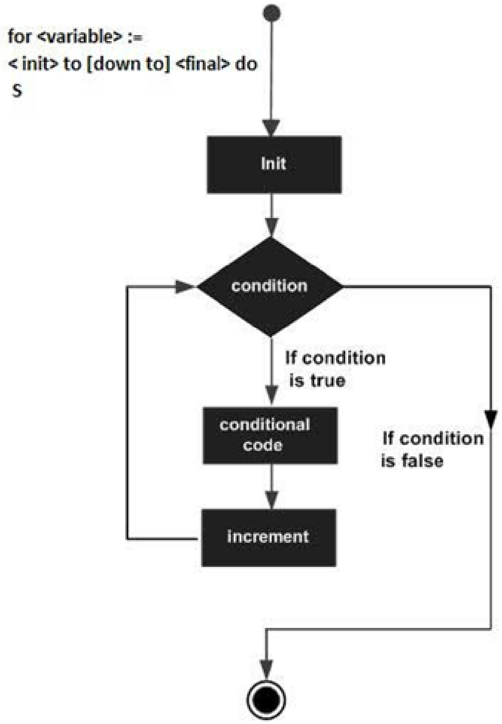
- Finite `for(i=0; i<10; i++)`

2. Logically-Controlled (condition-base)

- Finite `for (i=0; i<10; i++)`
- Indefinite `while (input != -1)`
- Infinite
  - `while(true)`
  - `while(i<10)`
  - `while('0')`
  - `for(;;){};`

Pascal `for` statement

```
program forLoop;
var
   a: integer;

begin
   for a := 10  to 20 do

   begin
      writeln('value of a: ', a);
   end;
end
```



C `for` statement

```
for ([expr_1] ; [expr_2] ; [expr_3])
    statement
```

- The expressions can be whole statements or statement sequences, separated by commas
  - The value of a multiple-statement expression is the value of the last statement in the expression
  - If second expression is absent, it is an infinite loop
- Design choices:
  - No explicit loop variable → the loop needs not count
  - Everything can be changed in the loop

```cpp
#include <iostream>
using namespace std;
int main(){
    int a;
    for(a = 10;;a++){ // infinite loop
        cout << "\n value of a: " << a;
    }
    for(a = 10;a<=20;a++){
        cout << "\n value of a: " << a;
    }
    return 0;
}
```

Which one is easier to read and write?

```pascal
program forLoop;
var
    a: integer;
begin
    for a := 10 to 20 do
    begin
        writeln('value of a: ', a);
    end;
end.
```

```cpp
#include <iostream>
using namepsace std;
int main(){
    int a;
    for(a = 10;a<=20;a++){
        cout << "\n value of a: " << a;
    }
    return 0;
}
```

Which one is easier to read and write?

```pascal
program forLoopbreak;
var
   a: integer;
   b: integer;
   function random():integer;
   begin
    random := 15;
   end;
begin
   b:= random();
   for a := 10  to 20 do
   begin
        if( a > b) then
        break;

      writeln(' do someting: ', a);
   end;

   writeln('exit value of a: ', a);
   writeln('random of b: ', b);
end.
```

```cpp
#include <iostream>

using namespace std;
int xrandom() {
    return 15;
}
int main()
{
    int b=xrandom();
    int a;
    for(a = 10 ; a < 20 && a <= b ; a++) {
        cout << "\n do someting:" << a ;
    }
    cout << "\n exit value of a: " << a ;
    cout << "\n random of b: " << b ;

    return 0;
}
```

C++ `for` statement

```
for (count1 = 0, count2 = 1.0;
     count1 <= 10 && count2 <= 100.0;
     sum = ++count1 + count2, count2 *= 2);
```

- C++ differs from earlier C in two ways:
  - The control expression can also be Boolean
  - Initial expression can include variable definitions (scope is from the definition to the end of loop body)
- Java and C#
  - Differs from C++ in that the control expression must be Boolean

What’s the result?

```
#include <iostream>

using namespace std;
int xrandom() {
    return 15;
}
int main()
{
    int b=xrandom();
    int a;
    for(a = 10 ; a ; a++) {
        cout << "\n do someting:" << a ;
    }
    cout << "\n exit value of a: " << a ;
    cout << "\n random of b: " << b ;

    return 0;
}
```

Python `for` statement

```
for loop_variable in object:
    # loop body
[else:
    # else clause
]
```

Case 1

```
for x in range(2):
  print(x)
else:
  print("Finally finished!")

print("end loop")
```

Case 2

```
for x in range(6):
  if x == 3:
    break
  print(x)
else:
  print("Finally finished!")
```

Swift `for` statement

Case 1

```swift
for i in 1...3  {
    print(i)
}
```

Case 2

```swift
print("Players gonna ")
 for _ in 1...5 {
    print("play")
}
```

Case 3

```swift
for i in 1...5  where i == 2 || i == 3 {
    print(i)
}
```

Counter-Controlled Loops in Functional Languages

- In imperative languages use a counter variable to keep state.
- But the functional languages uses recursion function with parameter to keep state.

https://shorturl.at/ZH7zA

```
let rec forLoop loopBody reps =
    if reps <= 0 then
        ()
    else
        loopBody()
        forLoop loopBody (reps - 1)

let printHi() = printf "Hi! "
forLoop printHi 3
```

Logically-Controlled Loops

- Repetition control based on Boolean expression
- C and C++ have both pretest and posttest forms, and control expression can be arithmetic:
- Java is like C, except control expression must be Boolean (and the body can only be entered at the beginning – Java has no `goto`)

```
while(ctrl_expr)
  loop body
```

```
do
  loop body
while (ctrl_expr)
```

Example

```
using System;
class HelloWorld {
  static void Main() {
    int sum = 0;
    int indat = Int32.Parse(Console.ReadLine());
    while (indat >= 0){
      sum += indat;
      indat = Int32.Parse(Console.ReadLine());
    }
  }
}
```

```
using System;
class HelloWorld {
  static void Main() {
    int value = Int32.Parse(Console.ReadLine());
    int digits = 0;
    do {
      value /= 10;
      digits++;
    } while(value>0);
  }
}
```

Example

https://onlinegdb.com/Fy7zRXrpB

```c
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <stdbool.h>
int main(){
  time_t currentTime;
  for(;;){
      time(&currentTime); // Get the current time
      printf("Current time: %s\n", ctime(&currentTime));

      sleep(1);
  }
  return 0;
}
```

https://onlinegdb.com/I9az9rvTP

```c
#include <stdio.h>
#include <unistd.h>
#include <time.h>
int main(){
  time_t currentTime;

  while(1){
      time(&currentTime); // Get the current time
      printf("Current time: %s\n", ctime(&currentTime));

      sleep(1);
  }
  return 0;
}
```

Example

https://onlinegdb.com/Pzz8P_sSa

```
#include <stdio.h>
int main(){
  int i=0;
  while(i<10){
      printf("%d\n",i);
      i++;
  }
  return 0;
}
```

https://onlinegdb.com/CGxf3tJ9y

```
#include <stdio.h>
int main(){
  for(int i=0;i<10;i++){
      printf("%d\n",i);
  }
  return 0;
}
```

Example

https://onlinegdb.com/skek30GIXC

```c
#include <stdio.h>
#define SIZE 10
int main(){
  int rulers[SIZE][SIZE]={0};
  for(int i=0,j=0;i<SIZE&&j<SIZE;
    j<SIZE-1?j++:j>=SIZE-1?j=0,i++:i){
      rulers[i][j]=j;
      printf("rulers[%d][%d] = %d\n",i,j,rulers[i][j]);
  }
  return 0;
}
```

https://onlinegdb.com/tYGWv8AX6

```c
#include <stdio.h>
#define SIZE 10
int main() {
    int i=0, j=0;
    int rulers[SIZE][SIZE]={0};
    while(i < SIZE) {
        rulers[i][j] = j;
        printf("rulers[%d][%d] = %d\n", i, j, rulers[i][j]);
        if(j < SIZE-1) {
            j++;
        } else {
            j = 0;
            i++;
        }
    }
    return 0;
}
```

Example

https://onlinegdb.com/FE0rpJyvM

```
#include <stdio.h>
#define SIZE 10
#define IS_FOR 1
int main(){
  int rulers[SIZE][SIZE]={0};
  #if IS_FOR == 1
  printf("For-Loop:\n");
  for(int i=0, j=0; i<SIZE; j<SIZE-1?j++:(j=0,i++)) {
      rulers[i][j] = j;
      printf("rulers[%d][%d] = %d\n", i, j, rulers[i][j]);
  }
  #else
  printf("While-Loop:\n");
  int i=0,j=0;
  while(i<SIZE){
      rulers[i][j] = j;
      printf("rulers[%d][%d] = %d\n", i, j, rulers[i][j]);
      j<SIZE-1?j++:(j=0,i++);
  }
  #endif
  return 0;
}
```

https://onlinegdb.com/ntPFafBC_

```
#include <stdio.h>
#define SIZE 10
#define IS_FOR 0
int main() {
    int rulers[SIZE][SIZE]= {0};
    #if IS_FOR == 1
    printf("For-Loop:\n");
    for(int k=0; k<SIZE*SIZE; k++) {
        int i = k / SIZE;  // Row
        int j = k % SIZE;  // Column
        rulers[i][j] = j;
        printf("rulers[%d][%d] = %d\n", i, j, rulers[i][j]);
    }
    #else
    printf("While-Loop:\n");
    for(int k=0; k<SIZE*SIZE; k++) {
        int i = k / SIZE;  // Row
        int j = k % SIZE;  // Column
        rulers[i][j] = j;
        printf("rulers[%d][%d] = %d\n", i, j, rulers[i][j]);
    }
    #endif
```

User-Located Loop Control Mechanisms

- Programmers decide a location for loop control (other than top or bottom of the loop)
- Simple design for single loops (e.g., `break`)
- C , C++, Python, Ruby, C# have unconditional unlabeled exits (`break`), and an unlabeled control statement, `continue` , that skips the remainder of current iteration, but not the loop
- Java and Perl have unconditional labeled exits (`break` in Java, `last` in Perl) and labeled versions of `continue`

break Statement

- Most languages. such as C/C++,java, the break statement terminates the loop when it is encountered.

```
for (init; condition; update) {
    // code
    if (condition to break) {
        break;
    }
    // code
}
```

```
while (condition) {
    // code
    if (condition to break) {
        break;
    }
    // code
}
```

```
for(i=0;true;i++){
  console.log(i);
  if(i<5){
    break;
  }
}
```

```
i=0;
while(true){
  console.log(i);
  i++;
  if(i<5){
    break;
  }
}
```

C++: break

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

int main() {
    int number;
    int sum = 0;
    while (true) {
        // take input from the user
        cout << "Enter a number: ";
        cin >> number;
        // break condition
        if (number < 0) {
            break;
        }
        // add all positive numbers
        sum += number;
    }
    // display the sum
    cout << "The sum is " << sum << endl;
    return 0;
}
```

Java: break

```java
public class Main
{
    public static void main(String[] args) {
          for (int i = 0; i < 10; i++) {
            // Terminate the loop when i is 5
            if (i == 5)
                break;
            System.out.println("i: " + i);
        }
        System.out.println("Out of Loop");
    }
}
```

Swift: Break label

```
outerloop: for i in 1...5{

  innerloop: for j in 1...5 {

    if j == 3 {
      print("before i = \(i), j = \(j)")
      break outerloop
    }

    print("i = \(i), j = \(j)")
  }
}
print("end ")
```

Java: Break label

```java
public class Main
{
    public static void main(String[] args)
    {
        int i=-1,j=-1;
        // First label
        first:
        for ( i = 0; i < 3; i++) {
        // Second label
        second:
            for ( j = 0; j < 3; j++) {
                if (i == 1 && j == 1) {

                    // Using break statement with label
                    System.out.println( "before break -> i = " +i + ", j = " + j);
                    break first;
                }
                System.out.println( "i = " +i + ", j = " + j);
            }
        }
        System.out.println( "after break -> i = " +i + ", j = " + j);
    }
}
```

Python: break

```
for x in range(2):
  print(x)
else:
  print("Finally finished!")

print("end loop")
```

```
for x in range(6):
  if x == 3: break
  print(x)
else:
  print("Finally finished!")
```

continue Statement

- The `continue` statement is used to skip the current iteration of the loop and the control of the program goes to the next iteration

```
for(i=0;i<10;i++){
  if(i%2==0){
    continue;
  }
}
```

```
i=0;
while(i<10){
  if(i%2==0){
    continue;
  }
}
```

continue in C/C++,C#

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

int main() {
    for (int i = 1; i <= 5; i++) {
        // condition to continue
        if (i == 3) {
            continue;
        }

        cout << i << endl;
    }

    return 0;
}
```

Iteration Based on Data Structures

- Number of elements in a data structure control loop iteration
- Control mechanism is a call to an iterator function that returns the next element in the data structure in some chosen order, if there is one; else loop is terminated
- C's for statement can be used to build a user-defined iterator:

```
for(p=root; p==NULL; traverse(p)){}
```

PHP vs. Java

```php
<?php
$list = array("apple", "banana", "cherry", "date");

reset($list);
print("1st: ".current($list)."<br />");
while($current_value = next($list))
    print("next: ".$current_value."<br />");
?>
```

```java
import java.util.ArrayList;
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        var tokens = new ArrayList<>(Arrays
        .asList("One", "Two", "Three"));
        for(String token: tokens) {
            System.out.println(token);
        }
    }
}
```

Python

```
fruits = ["apple", "cake", "banana", "cherry"]
count = 0;
for x in fruits:
  if x == "cake":
    continue
  print(x)
  count+=1
  if x == "banana":
    break
print (count);
```

C#

```
using System;
using System.Collections;
class HelloWorld {
  static void Main() {
        // adding elements using ArrayList.Add() method
        var arlist1 = new ArrayList();
        arlist1.Add(1);
        arlist1.Add("Bill");
        arlist1.Add(" ");
        arlist1.Add(true);
        arlist1.Add(4.5);
        arlist1.Add(null);
        foreach(var i in arlist1) {
            Console.WriteLine( i );
        }
  }
}
```

Swift

```swift
let languages = ["Swift", "Java", "Go", "JavaScript"]

for language in languages where language != "Java"{
  print(language)
}
```

JavaScript

Case 1

```
tokens = ["one","two","three"]

for(let token of tokens){
    console.log(token)
}
```

Case 2

```
const person = {fname:"John", lname:"Doe", age:25};
let text = "";
for (let x in person) {
  text += person[x] + " ";
}
```

Unconditional Branching (4.)

- Transfers execution control to a specified place in the program, e.g., goto
- Major concern: readability
  - Some languages do not support goto statement (e.g., Java)
  - C# offers goto statement (can be used in switch statements)
- Loop exit statements are restricted and somewhat hide away goto’s

https://onlinegdb.com/Hw2Ke1R85

```
#include<iostream>
using namespace std;
void checkGreater()
{
    int i, j;
    i=2;j=5;
    if(i>j)
        goto iGreater;
    else
        goto jGreater;
    iGreater:
        cout<<i<<"\n i is greater";
        goto end;
    jGreater:
    cout<<j<<"\n j is greater";
end:;
     cout<<"\n end" ;
    return;
}
int main()
{
    checkGreater();
    return 0;
}
```

```
goto label;
...
...
label:
...
...
```

```
label:
...
...
goto label;
...
...
```

Example goto

https://www.onlinegdb.com/Y-QG2OCki

```c
#include <stdio.h>

int main(){
    int i=0;
    int j=5;

    // Declare jump table BEFORE using it
    void *jumptable[2] = {&&end, &&loop};

loop:
    printf("i = %d\n", i);
    i++;

    // Use computed goto (GCC extension)
    int cond = (i < j);      // 1 if true, 0 if false
    goto *jumptable[cond];   // Jump to jumptable[0] or jumptable[1]

end:
    return 0;
}
```

https://onlinegdb.com/tZI0grDB5

```cpp
#include <iostream>
using namespace std;
int main()
{
    float num, average, sum = 0.0;
    int i, n;
    cout << "Maximum number of inputs: ";
    cin >> n;
    for(i = 1; i <= n; ++i)
    {
        cout << "Enter n" << i << ": ";
        cin >> num;
        if(num < 0.0)
        {
           // Control of the program move to jump:
            goto jump;
        }
        sum += num;
    }
jump:
    average = sum / (i - 1);
    cout << "\nAverage = " << average;
    return 0;
}
```

Controversy

history note

Although several thoughtful people had suggested them earlier, it was Edsger Dijkstra who gave the computing world the first widely read exposé on the dangers of the goto. In his letter he noted, "The goto statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program" (Dijkstra, 1968a). During the first few years after publication of Dijkstra's views on the goto, a large number of people argued publicly for either outright banishment or at least restrictions on the use of the goto. Among those who did not favor complete elimination was Donald Knuth (1974), who argued that there were occasions when the efficiency of the goto outweighed its harm to readability.

1. Dijkstra (1968) - Famously wrote "Go To Statement Considered Harmful," arguing that goto makes programs:

- Hard to follow logically
- Difficult to understand and debug
- Prone to creating "spaghetti code" (tangled, messy program flow)

2. Donald Knuth (1974) - Took a more nuanced view, arguing that:

- In some cases, goto can make code more efficient
- Complete elimination isn't always necessary
- Sometimes goto can actually be clearer than alternatives

```
else
{
    switch (text[12])
    {
    case '0':
        break;
    case '3':
        goto IL_013a;
    case '6':
        goto IL_0160;
    case '8':
        goto IL_0186;
    case '9':
        goto IL_01ac;
    default:
        goto IL_0239;
    }
    string text2 = text;
    string text3 = text2;
    if (!(text3 == "240232201896017305"))
    {
        if (!(text3 == "240232207254026098"))
        {
            goto IL_0239;
        }
        queueItemDG.Channel = "24";
    }
    else
    {
        queueItemDG.Channel = "23";
    }
}
goto IL_0245;
IL_013a:
if (text == "240232201199312025")
{
    queueItemDG.Channel = "26";
    goto IL_0245;
}
goto IL_0239;
IL_0160:
if (text == "240232207661699576")
{
    queueItemDG.Channel = "33";
    goto IL_0245;
}
goto IL_0239;
IL_0186:
if (text == "240232207264868502")
{
    queueItemDG.Channel = "25";
    goto IL_0245;
}
goto IL_0239;
IL_0239:
queueItemDG.Channel = "0";
goto IL_0245;
IL_01ac:
if (text == "240232202181978235")
{
    queueItemDG.Channel = "22";
```

https://onlinegdb.com/8MX4zIOgL

```
#include <iostream>
using namespace std;
int main()
{
    float num, average, sum = 0.0;
    int i, n;
    cout << "Maximum number of inputs: ";
    cin >> n;
    if( n > 1000)
     goto inloopfor;
    for(i = 1; i <= n; ++i)
    {
    inloopfor :;
        cout << "Enter n" << i << ": ";
        cin >> num;
        if(num < 0.0)
        { goto jump; }
        sum += num;
    }
jump:
    average = sum / (i - 1);
    cout << "\nAverage = " << average;
    return 0;
}
```

- Uncomment Goto inloopfor & inloopfor label https://onlinegdb.com/N7MwKb4xh

```
using System;
class HelloWorld {
  static void Main() {
   float num=0; float average =0; float sum = 0;
    int i, n;
    Console.WriteLine ( "Maximum number of inputs: ");
    n = Convert.ToInt32(Console.ReadLine());
    if( n > 1000) {
     // goto inloopfor;
      Console.WriteLine  ( "n > 1000 ");
    }
    for(i = 1; i <= n; ++i)
    {
        //inloopfor :;
         Console.WriteLine  ( "Enter n : " );
        n = Convert.ToInt32(Console.ReadLine());
        if(num < 0.0)
        { goto jump; }
        sum += num;
    }
jump:     ;
    average = sum / (i - 1);
  }
}
```

Guarded Commands (5.)

- Designed by Dijkstra
- Purpose: to support a new programming methodology that supports verification (correctness) during development
- Basis for two linguistic mechanisms for concurrent programming (in CSP and Ada)
- Basic Idea: if the order of evaluation is not important, the program should not specify one

Selection Guarded Command

```
if <Boolean exp> -> <statement>
[] <Boolean exp> -> <statement>
 ...
[] <Boolean exp> -> <statement>
fi
```

- Semantics:
  - Evaluate all Boolean expressions
  - If > 1 are true, choose one non-deterministically
  - If none are true, it is a runtime error
  - Prog correctness cannot depend on statement chosen

- Key Difference

```
if x > 0 then
  y := y + 1
elif x = 0 then
  y := 0
else
  y := y - 1
end if
```

```
if x > 0 -> y := y + 1
[] x = 0 -> y := 0
[] x < 0 -> y := y - 1
```

- ถ้ามีหลาย Guard และมี Boolean Expression มีค่าเป็นจริงพร้อมกัน จะสามารถเลือกได้ทุก Path

```
if buffer_not_full -> produce()
[] buffer_not_empty -> consume()
fi
```

- ถ้า buffer ทั้ง **ไม่เต็มและไม่ว่าง** → เลือกได้ทั้ง `produce()` หรือ `consume()`
- ไม่ต้องกำหนดว่าต้องเลือก `produce()` ก่อน `consume()` เสมอ → ลด bias ในการออกแบบ

https://onlinegdb.com/f1HDjuYLSr

```
#include <iostream>
#include <unistd.h>
#define MAX_BUFFER 6;
using namespace std;
int i = 0;
bool bufferNotFull(){return i < MAX_BUFFER;}
bool bufferNotEmpty(){return i > 0;}
void produce() {
    cout << "produce: " << ++i << "\n";
}
void consume(){
    cout << "consume: " << --i << "\n";
}
int main()
{
    while(true){
        if(bufferNotFull()){
            produce();
        }
        if(bufferNotEmpty()){
            consume();
        }
        sleep(3);
    }
    return 0;
}
```

Example

https://onlinegdb.com/nKC6ffutN

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main() {
    int x, y = 0;
    printf("Enter x: ");
    scanf("%d", &x);

    int choice = -1;
    if (x > 0 && x == 0 && x < 0) {
        // logically impossible all at once
    }
```

```c
    // Determine which guards are true
    int guards[3], count = 0;
    if (x > 0) guards[count++] = 1;
    if (x == 0) guards[count++] = 2;
    if (x < 0) guards[count++] = 3;

    srand(time(NULL));
    if (count > 0) {
        choice = guards[rand() % count];
        switch(choice) {
            case 1: y = y + 1; break;
            case 2: y = 0; break;
            case 3: y = y - 1; break;
        }
    }
    printf("y = %d\n", y);
    return 0;
}
```

Loop Guarded Command

```
do <Boolean> -> <statement>
[] <Boolean> -> <statement>
 ...
[] <Boolean> -> <statement>
od
```

- Semantics: for each iteration
  - Evaluate all Boolean expressions
  - If more than one are true, choose one non-deterministically; then start loop again
  - If none are true, exit loop

Conclusions (6.)

- Variety of statement-level structures
- Choice of control statements beyond selection and logical pretest loops is a trade-off between language size and writability
- Functional and logic programming languages are quite different control structures

Homework

- Rewrite the program with C#/Java

5. In a letter to the editor of *CACM*, Rubin (1987) uses the following code segment as evidence that the readability of some code with gotos is better than the equivalent code without gotos. This code finds the first row of an *n* by *n* integer matrix named x that has nothing but zero values.

```
for (i = 1; i <= n; i++) {
  for (j = 1; j <= n; j++)
     if (x[i][j] != 0)
      goto reject;
  println ('First all-zero row is:', i);
  break;
reject:
 }
```