

03: CONTROL STRUCTURES

Programming Technique I
(SECJ1013)



Dr. Alif Ridzuan Bin Khairuddin
Senior Lecturer
Department of Software Engineering
Faculty of Computing
Universiti Teknologi Malaysia

**FOR MID SEMESTER
TEST USE**

Boolean and Logical Operator

- In C++ logical data declared as **bool** data type
e.g.

```
bool variable_name;
```

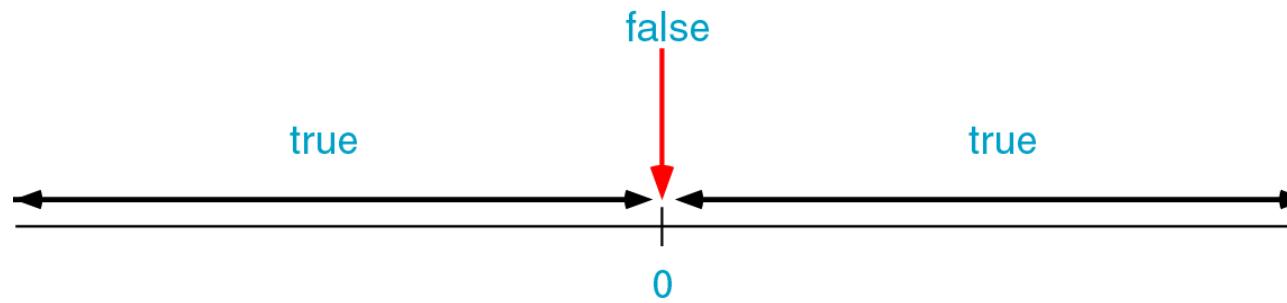
- There are only two values: **true** and **false**
- Type-casting **bool** to **int**:
 - **true** => 1
 - **false** => 0

Example

```
int number;
number = 2 + true;
cout << number; //output: 3
```

Boolean and Logical Operator

- Type-casting `int` to `bool`:
 - A *Zero value* => `false`
 - A *Non-Zero value* => `true`



Example:

```
bool b = false;           // b initially is false
int number = 0;
b = -10;                  // Now, b is true
b = number;                // Here, b is false again
```

Boolean and Logical Operator

What would be printed by this code segment

```
bool b;  
int p;  
int q = 5;  
  
b = q;  
p = b;  
cout << "The value of p is " << p << endl;
```

Logical operators truth table

not

x	!x
false	true
true	false

!

x	!x
zero	1
nonzero	0

logical

and

x	y	x&&y
false	false	false
false	true	false
true	false	false
true	true	true

logical

&&

x	y	x&&y
zero	zero	0
zero	nonzero	0
nonzero	zero	0
nonzero	nonzero	1

C Language

||

x	y	x y
false	false	false
false	true	true
true	false	true
true	true	true

logical

||

x	y	x y
zero	zero	0
zero	nonzero	1
nonzero	zero	1
nonzero	nonzero	1

C Language

Operations for logical and/or

false && (anything)



false

true || (anything)



true

Relational operators

Operator	Meaning
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	equal
!=	not equal

Logical expression

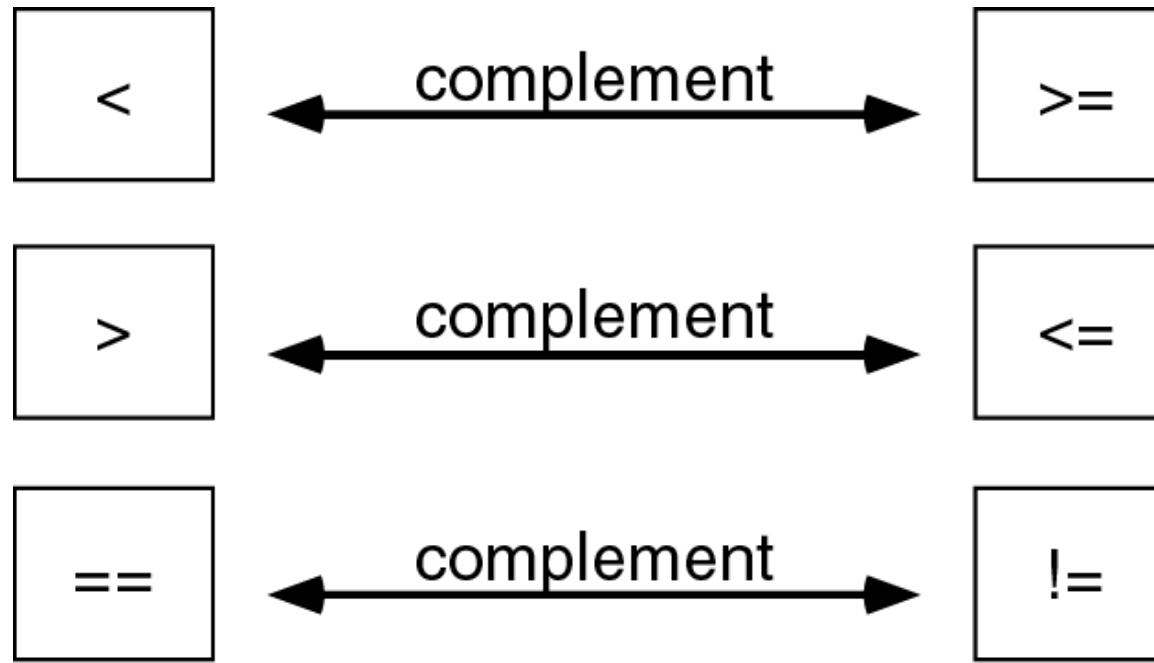
Example:

```
int a=10;

cout << a;
cout << (a==1);
cout << (a>1);
cout << (a=5);

a = (a != 5);
out << a;
```

Logical operator complements

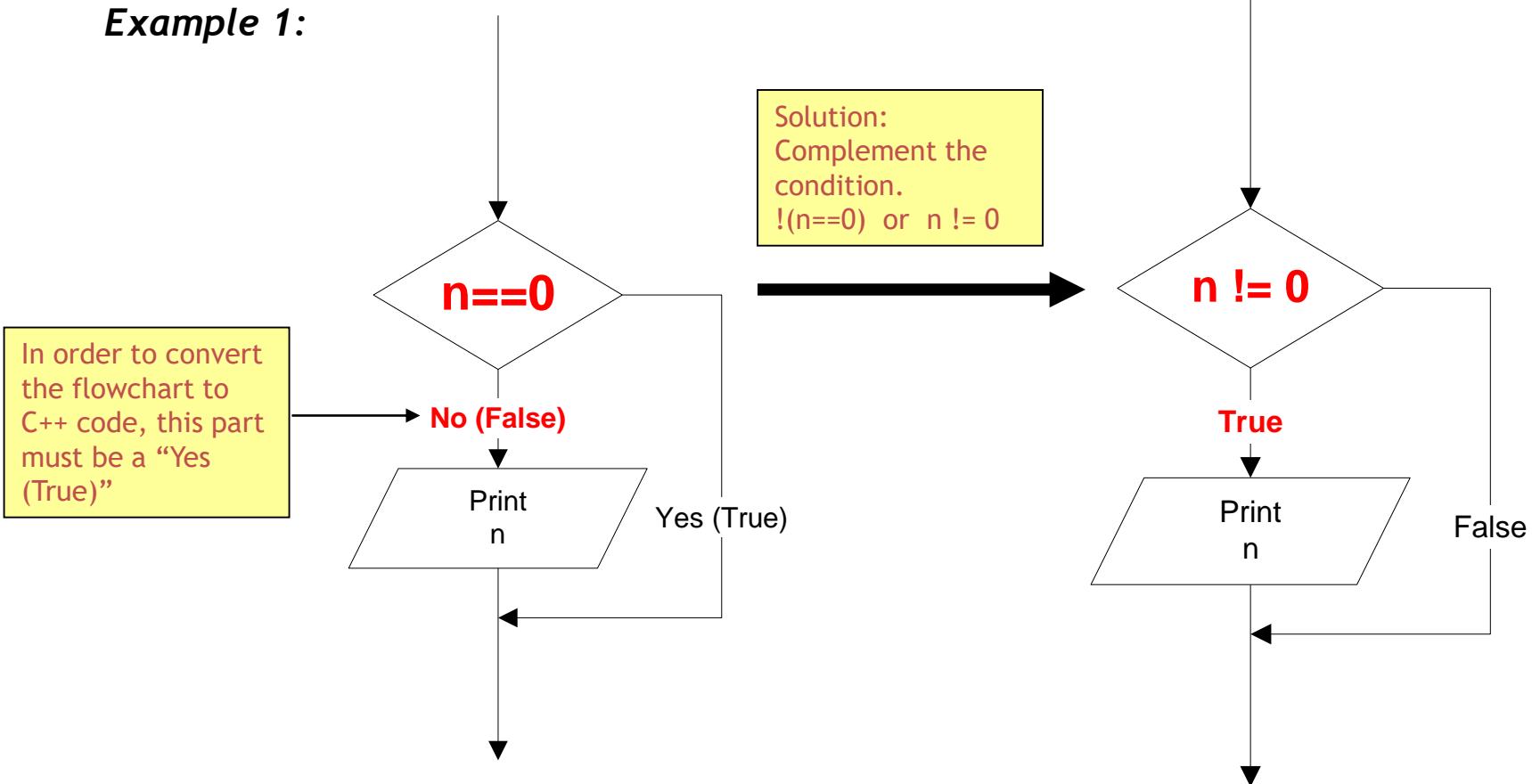


- Another way to complement an expression is just putting a Not operator (!) in front of it.

Example: Complement of $n==0$ is
 $! (n==0)$

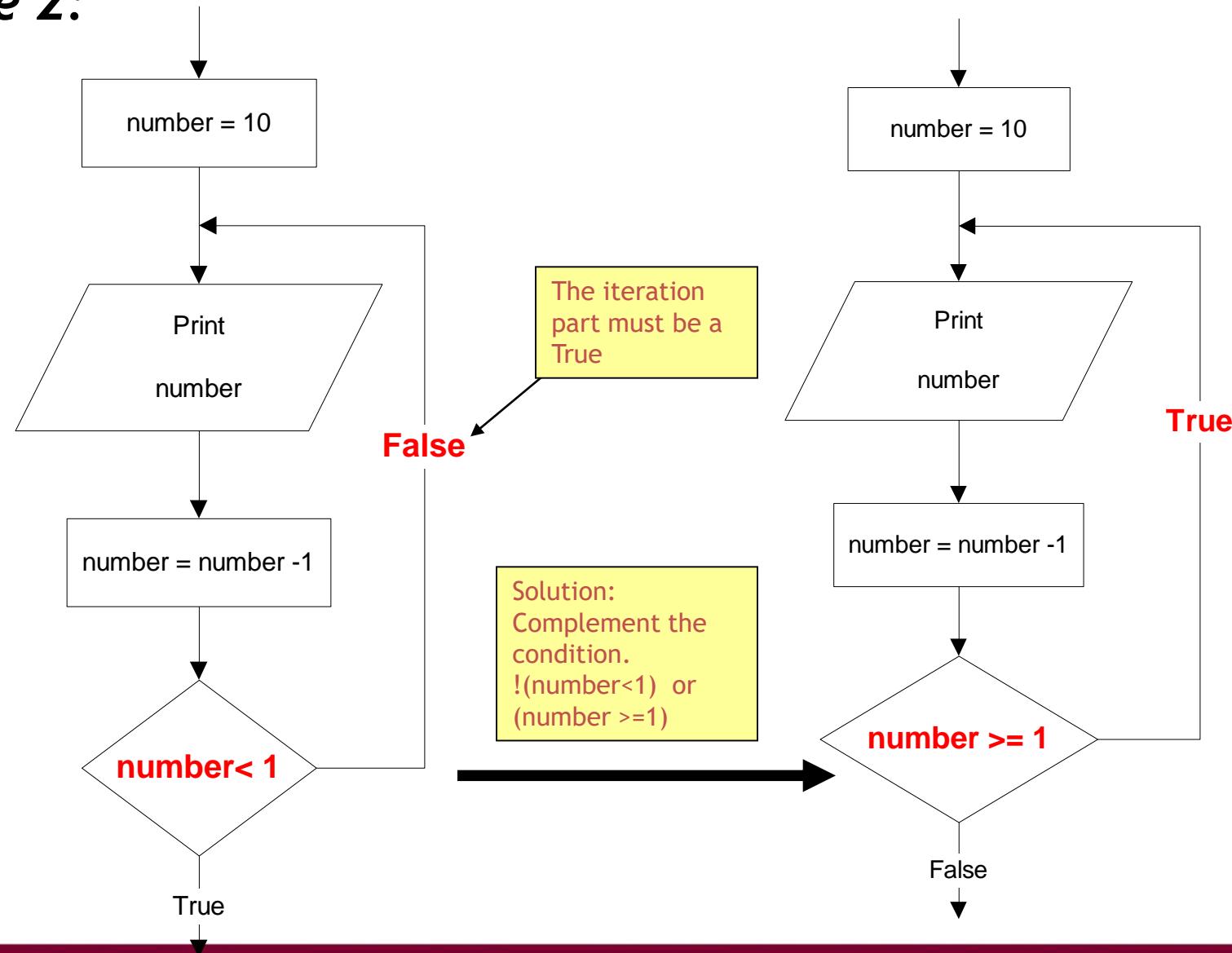
- When to use complement?

Example 1:



When to use complement?

Example 2:



Selection / Branch

- Sometimes your programs need to make logical choices.
- Example:

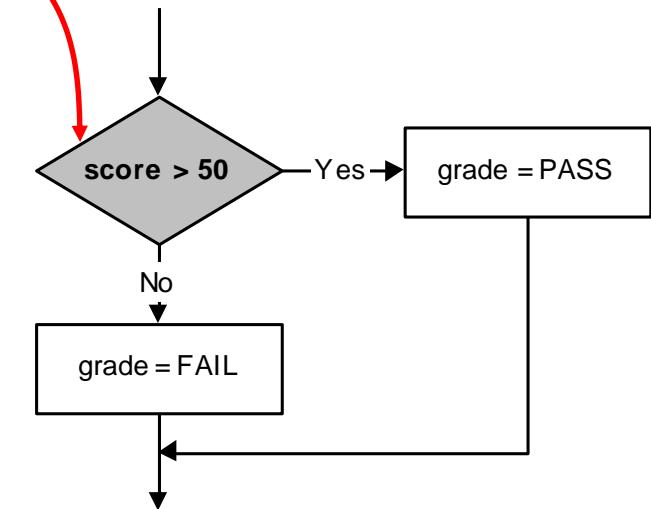
IF score is higher than 50
THEN grade is PASS
ELSE grade is FAIL
- In C++, this corresponds to **if** statement with three parts:

```
if (score > 50)    //part 1
{
    grade = PASS;   //part 2
}
else
{
    grade = FAIL;   //part 3
}
```

if statement

- Part 1 : the **condition** - an expression that evaluates to **true** or **false**.

```
if (score > 50)
{
    grade = PASS;
}
else
{
    grade = FAIL;
}
```



if statement

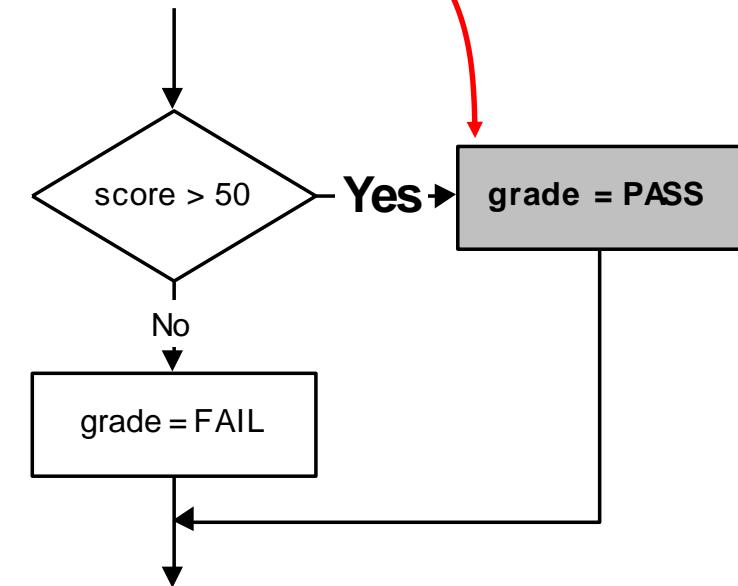
- Part 2 : the **TRUE-PART** - a block of statements that are executed if the condition evaluates to **true**

```

if (score > 50)
{
    grade = PASS;
}

else
{
    grade = FAIL;
}

```



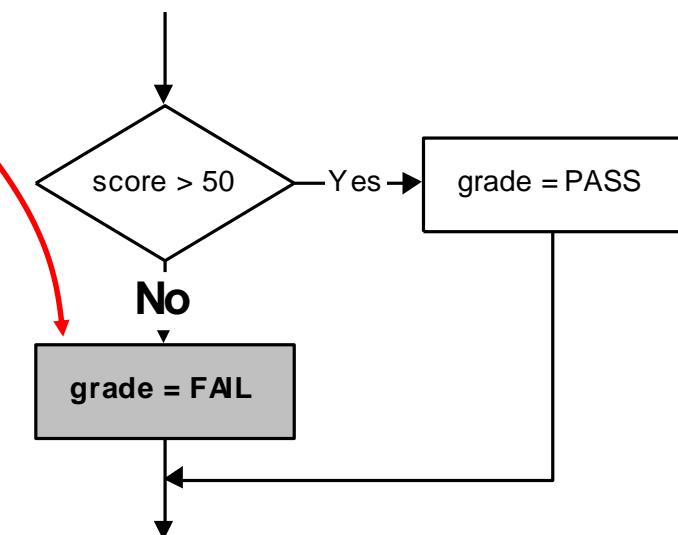
if statement

- Part 3 : the **FALSE-PART**- a block of statements that are executed if the condition evaluates to **false**

```

if (score > 50)
{
    grade = PASS;
}
else
{
    grade = FAIL;
}

```

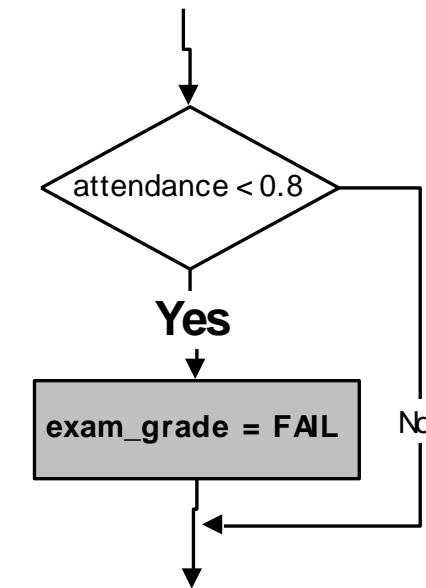


if the condition
evaluates to **false**,
the **TRUE-PART** is skipped.

if statement

- Sometimes there is no FALSE-PART. The “else” is omitted

```
if ( attendance < 0.8 )  
{  
    exam_grade = FAIL;  
}
```



if statement

- If the TRUE-PART (or FALSE-PART) consists of only **one statement**, then the curly braces may be omitted.
- *Example: these two statements are equivalent:*

```
if (score > 50)
{
    grade = PASS;
}
else
{
    grade = FAIL;
}
```

```
if (score > 50)
    grade = PASS;
else
    grade = FAIL;
```

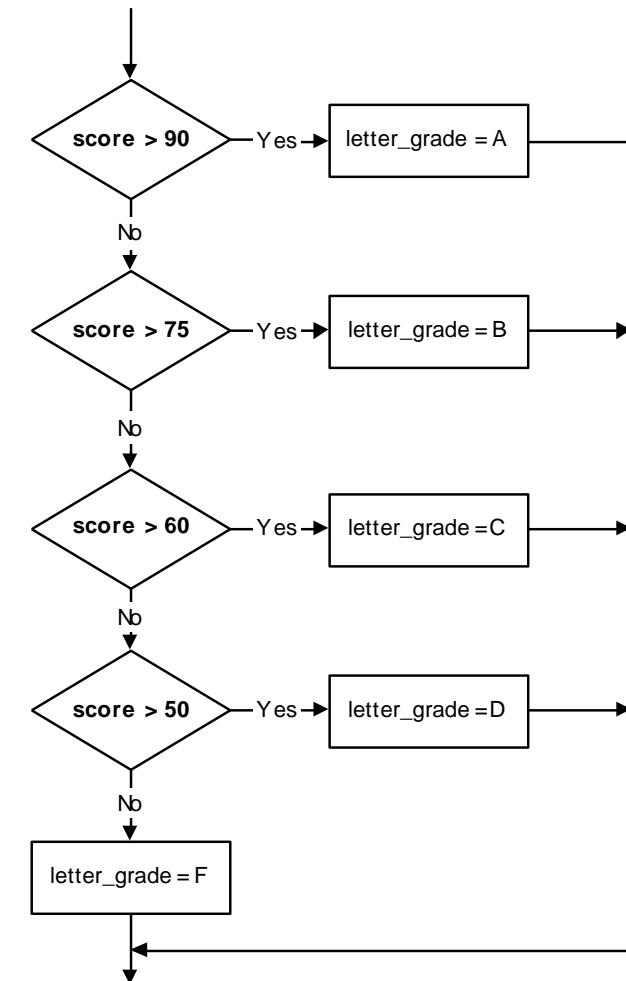
if statement

- Sometimes there are more than two parts. In those cases you may use **nested if-else** statements:

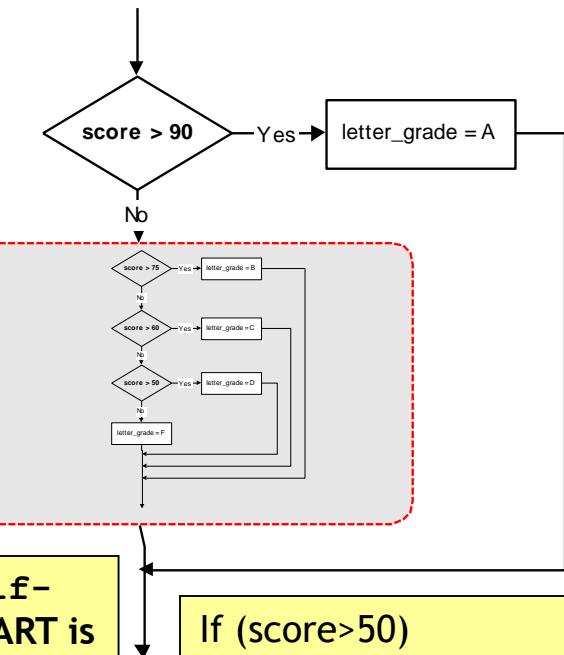
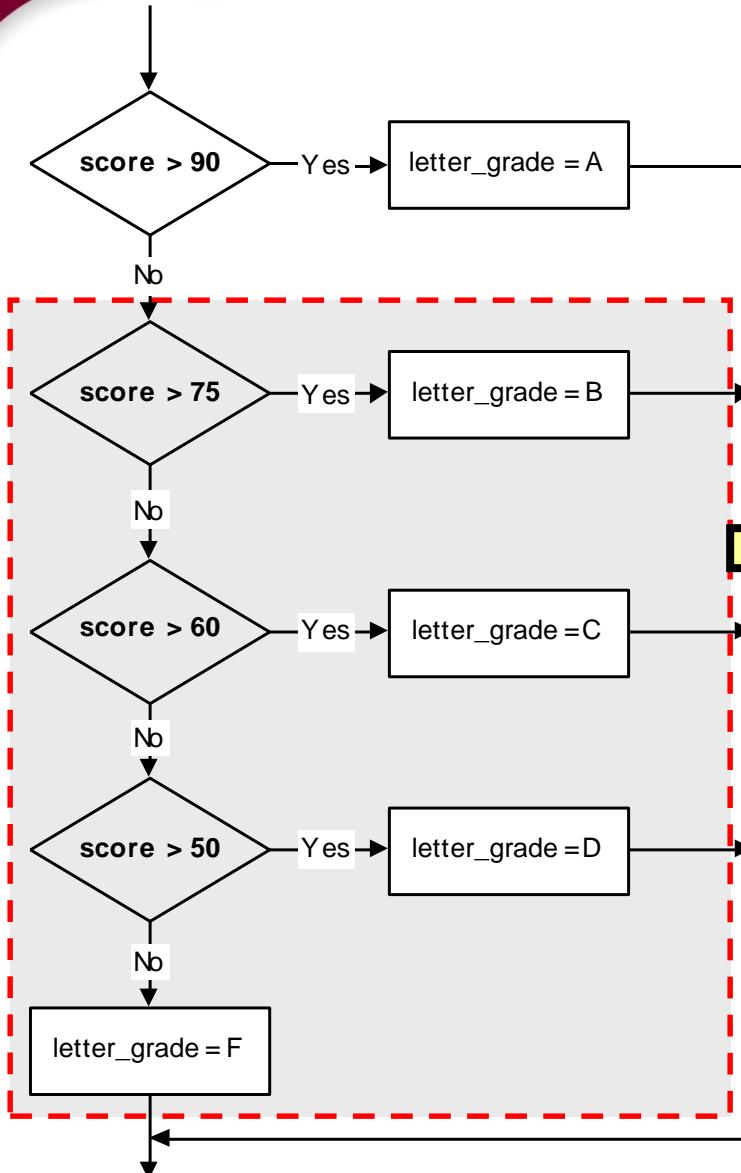
```

if (score > 90)
    letter_grade = 'A';
else if (score > 75)
    letter_grade = 'B';
else if (score > 60)
    letter_grade = 'C';
else if (score > 50)
    letter_grade = 'D';
else
    letter_grade = 'F';

```



Let's look closer



It is actually a regular **if-else** with the FALSE-PART is another **if-else** statement

```

If (score>50)
{
  letter_grade = 'A';
}
else {
  .....
  .....
}
  
```

if statement

- Three forms of **if** statements are shown at the next table.
- The *condition must be placed in parentheses*
- Statement may exist either as a single statement or as a collection of statements (also called **compound statement**)

```
if(condition)
    statement;
```

```
if (condition)
{   statement;
    |
    statement;
}
```

```
if (condition)
{   statement;
    |
    statement;
}
else
{   statement;
    |
    statement;
}
```

- A **compound statement** is one or more statements that are grouped together by enclosing them in brackets , **{ }**.
- Example:

```
if (value > 0)
    cout << value; ← This is a single statement. The
                        semi-colon belongs to "if" not
                        to "cout"

value = value * 2; ← a single statement

if (value > 10)

{
    value = 10;
    cout << value;

}
```

This is a compound statement which consists two single statements.

Related issues

- The condition must be placed in parentheses

Example:

```
if (0<x) && (x<10)      //syntax error
    cout << x;
```

Correction:

Related issues

- But be careful when converting mathematical comparisons. Some of them are not straight forward

Example: Print x only if (2<x<9)

```
if (2 < x < 9)  
cout << x;
```

There is no syntax error, but this leads to a **logic error** due to the misinterpretation.

The condition **always evaluates to true**, whatever the value of x

Let say x=1
 $(2 < x < 9)$
 $\Rightarrow (2 < 1 < 9)$
 $\Rightarrow (\text{false} < 9)$
 $\Rightarrow (0 < 9)$
 $\Rightarrow \text{true}$

Let say x=5
 $(2 < x < 9)$
 $\Rightarrow (2 < 5 < 9)$
 $\Rightarrow (\text{true} < 9)$
 $\Rightarrow (1 < 9)$
 $\Rightarrow \text{true}$

Correction:

Related issues

- The condition must evaluate to a Boolean value (i.e. either **true** or **false**)
- There are only two types of expression that result a Boolean value
 - Comparison expression (e.g. **a>2**)
 - Boolean expression (e.g. **b && false**)
- If the result of the condition is not a Boolean, it will be type-casted

Example:

The condition evaluates to
0. It then is type-casted to
Boolean, becomes **false**

```
int n=0;  
  
if (n)  
    cout << "Yes";  
else  
    cout << "No";
```

Output:

Example:

The condition evaluates to
5. It then is type-casted to Boolean, becomes **true**

```
int n=0;  
  
if (n + 5)  
    cout << "Yes";  
else  
    cout << "No";
```

Output:

Example:

Remember! This is an **assignment expression**, not an equality.

The value of the expression is 0. It then is type-casted to Boolean, becomes **false**. The result is always false.

```
int x=0;  
  
if (x=0)  
    cout << "Yes";  
else  
    cout << "No";
```

Output:

Example:

Remember! This is an **assignment expression**, not an equality.

The value of the expression is **10**. It then is type-casted to Boolean, becomes **true**. The result is always true.

```
int y=5;  
  
if (y=10)  
    cout << "Yes";  
else  
    cout << "No";
```

Output:

Example:

Remember! This is an **assignment expression**.

The condition always evaluates to **true**.
The value of **y** is changed to 5 due to the side-effect caused by the assignment operator

```
int y=1;  
  
if (y=5)  
    cout << y;
```

Output:

Related issues

- Be careful when using the Boolean operator NOT (**!**)

Example:

```
int n=5;  
  
if (!n>9)  
    cout << "Yes";  
else  
    cout << "No";
```

Operator **!** has higher precedence than operator **>**. So, it is executed first.

Expression **!n** is evaluated as **!true** where **n** is type-casted from integer 5 to Boolean true. The result is **false**

The expression is further evaluated as **(false>9)**. The **false** value is then type-casted to 0, since it will be compared with an integer. The expression then looks like **(0 > 9)** and the final result is **false**

Output:

Example:

```
int n=5;  
  
if (! (n>9))  
    cout << "Yes";  
else  
    cout << "No";
```

```
(! (n>9))  
⇒ (! (n>9))  
⇒ (! (5>9))  
⇒ (! (false))  
⇒ (!false)  
⇒ true
```

Output:

Related issues

- Statements should be indented correctly to avoid misinterpretations

Example:

```
if (x<3)
    cout <<"Yes" << endl;
    cout <<"No" << endl;
```

The second `cout` doesn't belong to `if` statement. It is on its own but was indented incorrectly.

Let say $x=1$,
Condition => true

Output:

Let say $x=3$
Condition => false

Output:

Correction:

```
if (x<3)
    cout <<"Yes" << endl;

    cout <<"No" << endl;
```

Example:

```
if (x<y)
    cout << x;
    x = y;
else
    cout << y;
```

Syntax error - misplace else.

There must only be a single statement before **else**. If more than that, use a compound statement.

Correction:

Example:

Print x only if it is an odd number less than 10, otherwise print “Wrong number”

```
if (x%2==1)
    if (x<10)
        cout <<x;
else
    cout << "Wrong number";
```

There is no syntax error, but this leads to a **logic error** due to the misinterpretation.

The **else** part actually belongs to the **second if** (`if (x<10)`), not to the first one

Let say x=7,
Output:

7

Let say x=11,
Output:

Wrong Number

Correct!

Correct!

But, when x=12,
There is no output. This is **incorrect**.
It suppose to print “Wrong number”

Correction:

Related issues

- Null statements are statements that do nothing

Example:

```
if (x<3);  
    cout <<"Yes";
```

The semi-colon represents a null statement. Either the condition evaluates to true or false, there is nothing to do.

The `cout` doesn't belong to `if` statement. The statement has already been ended up with semi-colon previously.

Example:

```
if (x<3)
    cout <<"Yes" <<endl;
else;←
    cout <<"No" <<endl;
```

Let say x=5,
Output:

Let say x=1,
Output:

The semi-colon represents a null statement.

This `cout` doesn't belong to `else` part.

Simplifying if statements

- Simplifying conditions:

Original statement

```
if ( a != 0 )  
    statement;
```

```
if ( a > 0 )  
    statement;
```

```
if ( a < 0 )  
    statement;
```

Simplified statement

```
if ( a )  
    statement;
```

```
if ( a == 0 )  
    statement;
```

```
if ( !a )  
    statement;
```

Simplifying if statements

- Example 1 : print a number only if it is an **odd** number

Original statement

```
if ( n%2==1 )  
    cout << n;
```

Simplified statement

```
if ( n%2 )  
    cout << n;
```

- Example 2: print a number only if it is an **even** number

Original statement

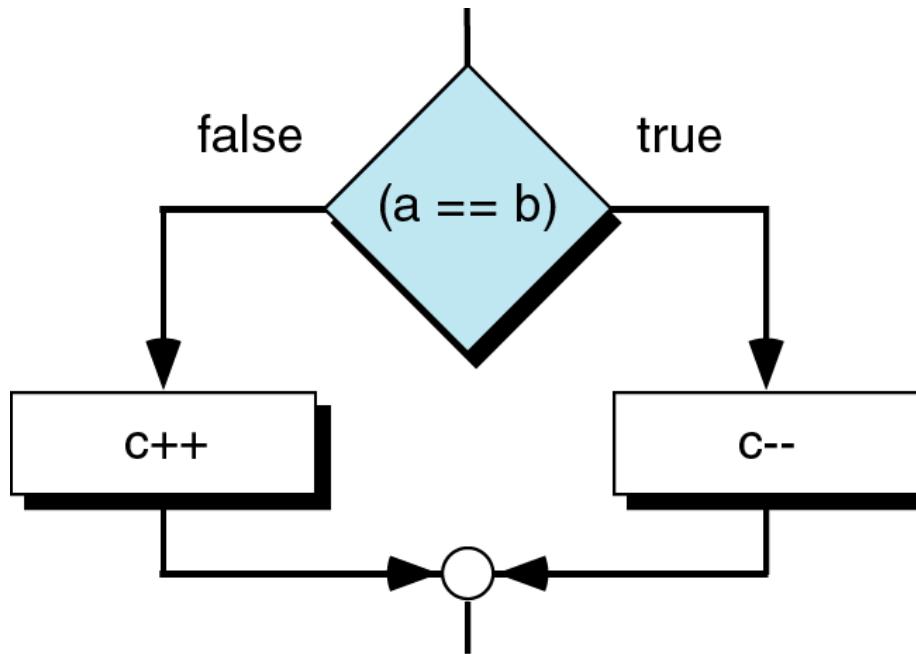
```
if ( n%2==0 )  
    cout << n;
```

Simplified statement

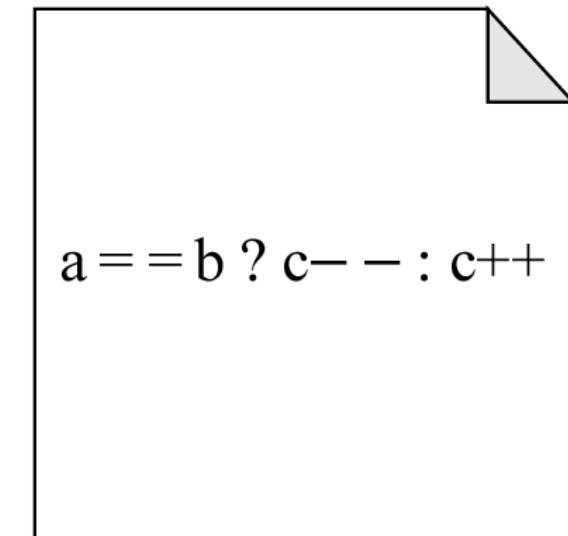
```
if ( !(n%2) )  
    cout << n;
```

Simplifying if statements

- Conditional Expressions:



(a) Logic Flow



(b) Code

Simplifying if statements

- Conditional Expressions:

Syntax: `condition ? value1 : value2`

If the condition is **true**, take the **value1**

If the condition is **false**, take the **value2**

Example:

`p = (p<5) ? q + 1 : 5;`

This statement means

```
if (p<5)
    p = q + 1;
else
    p = 5;
```

switch statement

- If there are many nested if/else statements, you may be able to replace them with a switch statement:

```
if (letter_grade == 'A')
    cout << "Excellent!";
else if (letter_grade == 'B')
    cout << "Very good!";
else if (letter_grade == 'C')
    cout << "Good";
else if (letter_grade == 'D')
    cout << "Adequate";
else
    cout << "Fail";
```



```
switch (letter_grade)
{
    case 'A' : cout <<"Excellent!";
                 break;

    case 'B' : cout <<"Very good!";
                 break;

    case 'C' : cout <<"Good";
                 break;

    case 'D' : cout <<"Adequate";
                 break;

    default   : cout <<"Fail";
                 break;
}
```

switch statement

```
switch (expression)
{
    case value1: statements_1;
                    break;

    case value2 : statements_2;
                    break;

    ...
    default : statements;
                break;
}
```

How the **switch** statement works?

1. Check the value of **expression**.
2. Is it equal to **value1**?
 - If yes, execute the **statements_1** and **break** out of the switch.
 - If no, is it equal to **value2**? etc.
3. If it is not equal to any values of the above, execute the **default statements** and then **break** out of the switch.

switch statement

Example 1:

```
int value = 1;  
  
switch (value)  
{  
    case 1: cout << "One"; ← Prints One  
    break;  
  
    case 2: cout << "Two";  
    break;  
  
    default : cout << "Neither One nor Two";  
    break;  
}
```

it is equal to this case-value (i.e. $1==1$). So, execute the statements of 'case 1'.

evaluates to 1

break out of the switch

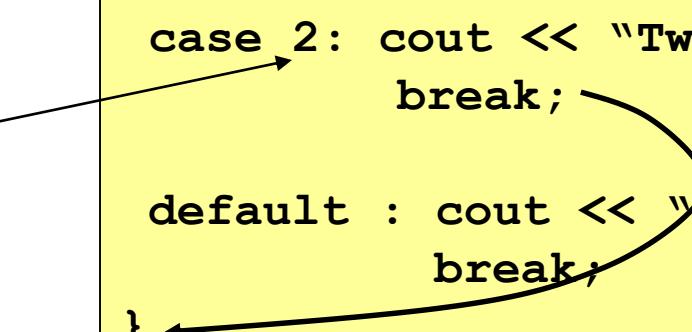
switch statement

Example 2:

it is not equal to this case-value (i.e. $2 \neq 1$). So, skip the statements of 'case 1' and move to the next case.

it is equal to this case-value (i.e. $2 == 2$). So, execute the statements of 'case 2'.

```
int value = 1;           this expression  
                        evaluates to 2  
  
switch (value + 1)  
{  
    case 1: cout << "One";  
             break;  
  
    case 2: cout << "Two";  Prints Two  
             break;  
  
    default : cout << "Neither One nor Two";  
              break;  
}
```



switch statement

Example 3:

The switch expression (i.e. 5) is not equal to both cases (i.e. $5!=1$ and $5!=2$). So, their statements are skipped.

When the 'default case' is reached, its statements are always executed.

```
int value = 5;
switch (value)
{
    case 1: cout << "One";
              break;

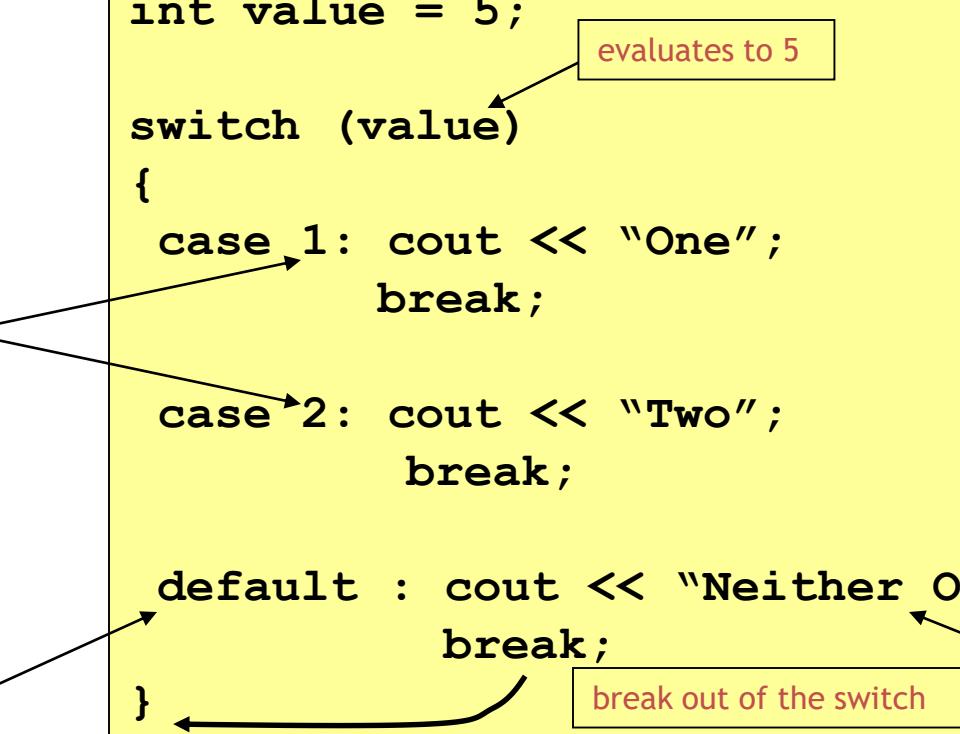
    case 2: cout << "Two";
              break;

    default : cout << "Neither One nor Two";
              break;
}
```

evaluates to 5

Prints Neither One nor Two

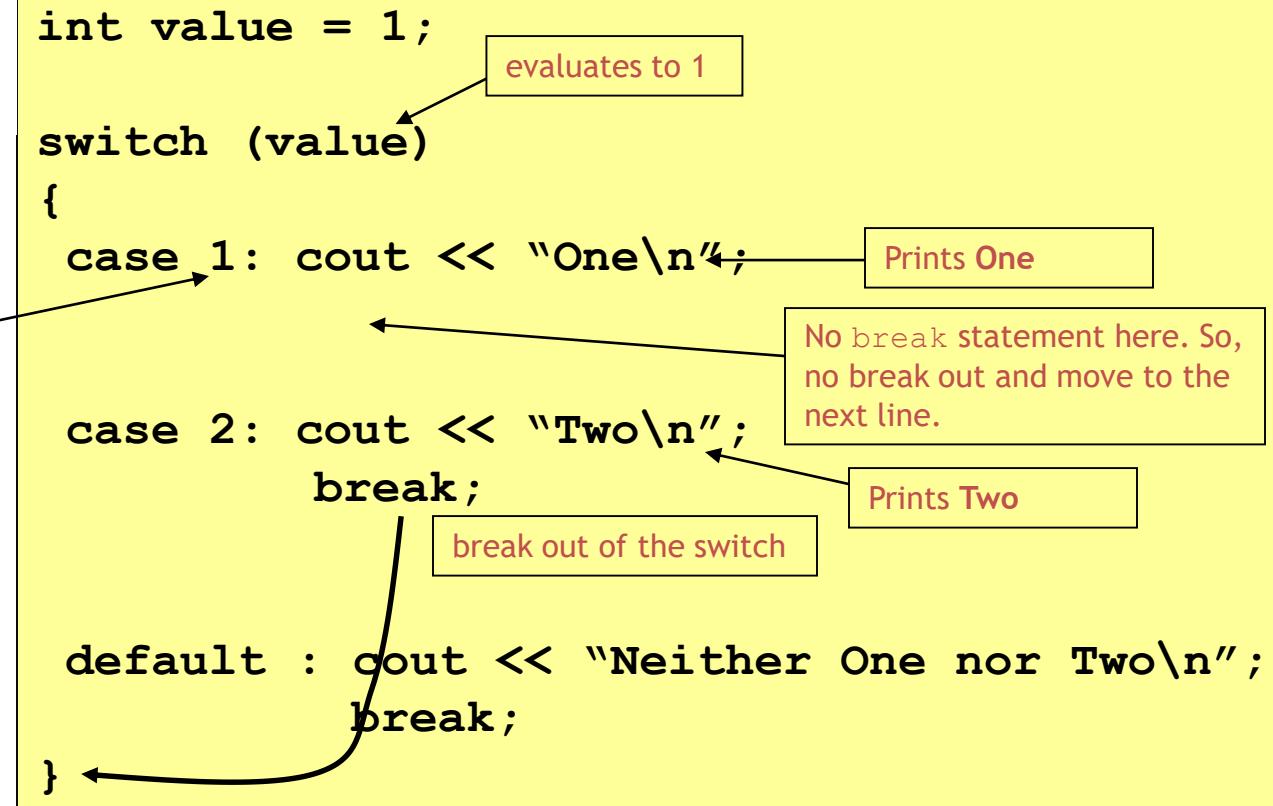
break out of the switch



switch statement

*What if the **break** statement is not written?*

it is equal to this case-value (i.e. `1==1`). So, execute the statements of the 'case 1'.



switch statement

- The switch expression must be of integral type (i.e. `int, char, bool`).
- The following examples would be an error

```
void main()
{
    float point=4.0;
    int mark;

    switch (point)
    {
        case 4      : mark = 100;
                        break;

        case 3.7   : mark = 80;
                        break;

        default   : mark = 0;
                        break;
    }
}
```

Error! The switch expression cannot be a `float` value

```
void main()
{
    char name []="Ali";
    int mark;

    switch (name)
    {
        case "Ali"     : mark=95;
                        break;

        case "Aminah": mark=90;
                        break;

        default       : mark=50;
                        break;
    }
}
```

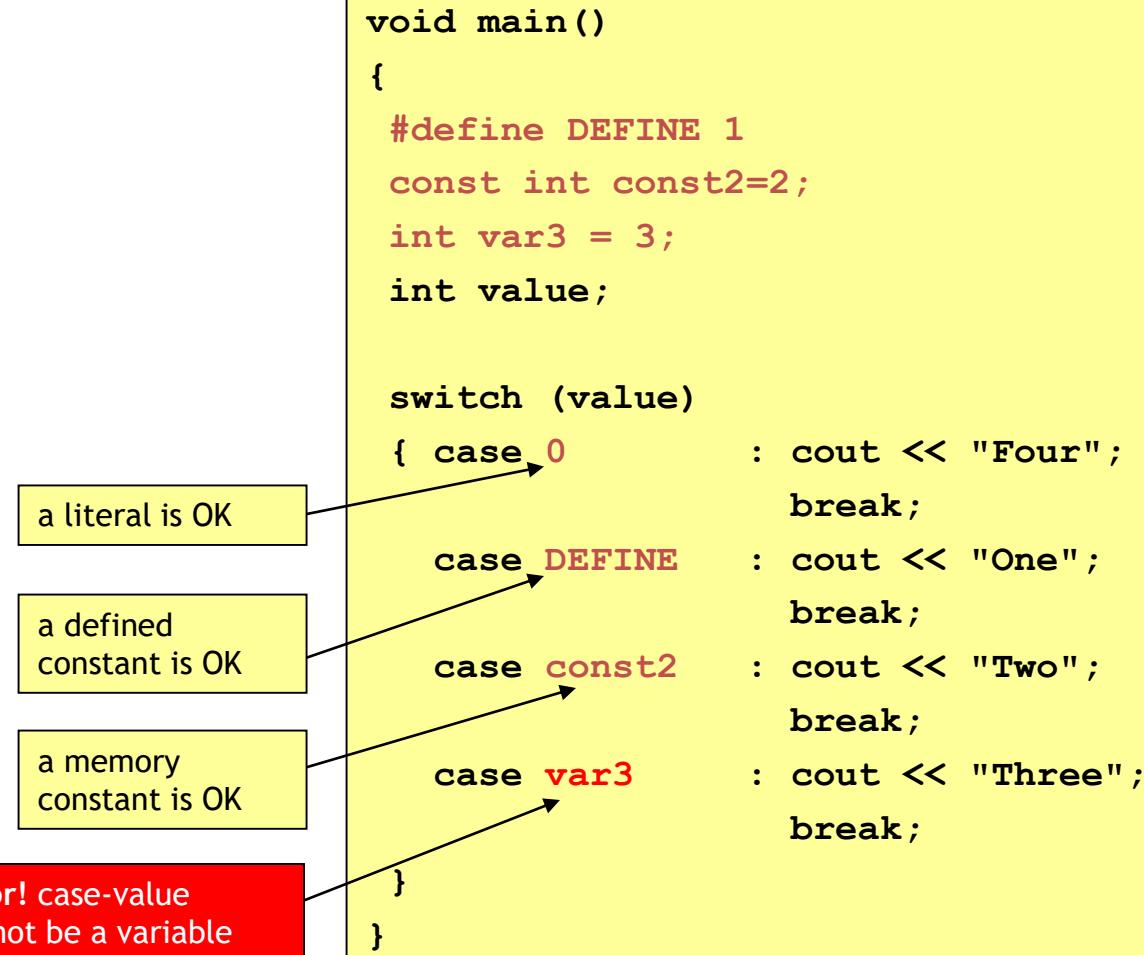
Error! The switch expression cannot be a `string` value

switch statement

- The case-value must be a constant (literal, memory or defined constant)
- The following example would be an error

```
void main()
{
    #define DEFINE 1
    const int const2=2;
    int var3 = 3;
    int value;

    switch (value)
    { case 0 : cout << "Four";
        break;
        case DEFINE : cout << "One";
        break;
        case const2 : cout << "Two";
        break;
        case var3 : cout << "Three";
        break;
    }
}
```

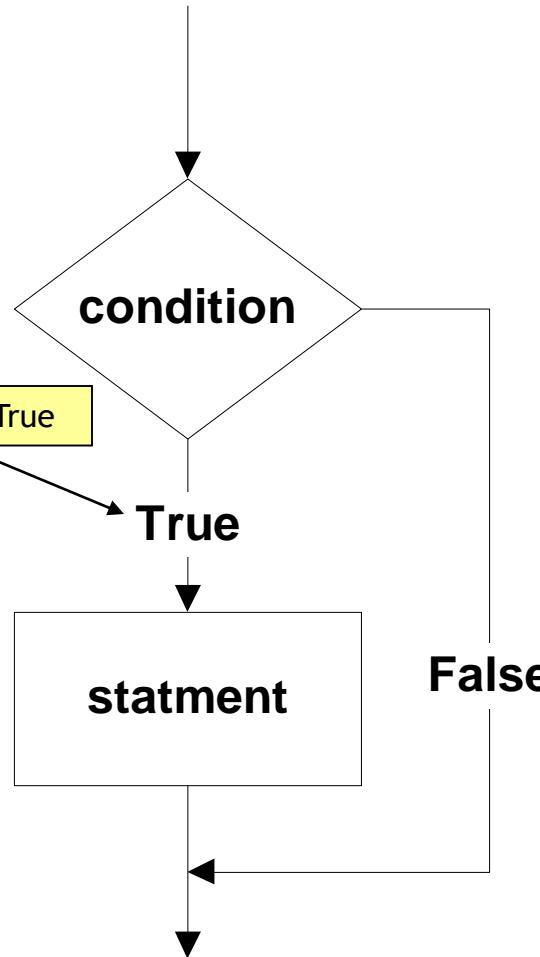


The diagram illustrates the annotations for the switch statement cases:

- a literal is OK**: Points to the first case value, `0`.
- a defined constant is OK**: Points to the second case value, `DEFINE`.
- a memory constant is OK**: Points to the third case value, `const2`.
- Error! case-value cannot be a variable**: Points to the fourth case value, `var3`.

Translating flowchart to C++ code

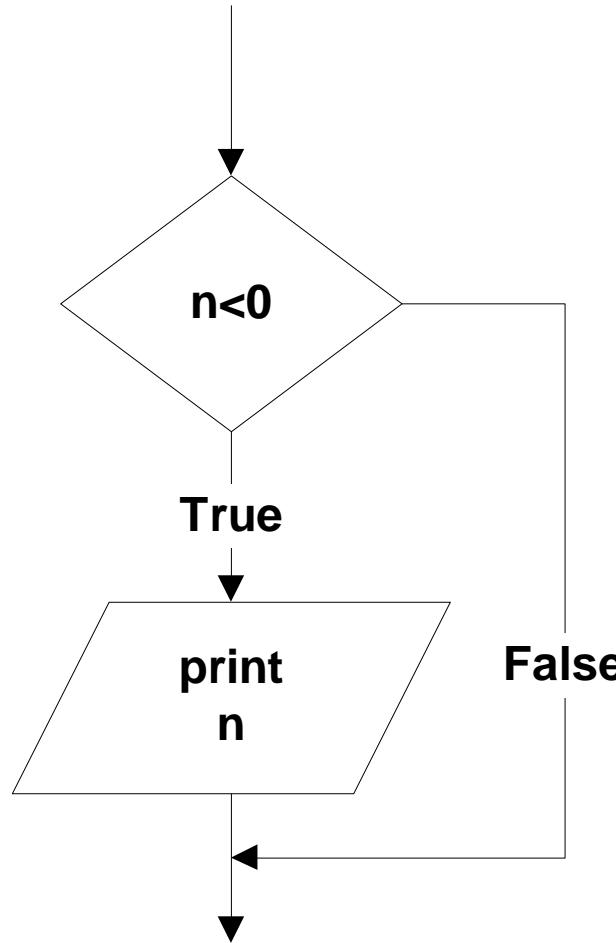
Pattern 1



```
if (condition)
{
    statement;
}
```

Translating flowchart to C++ code

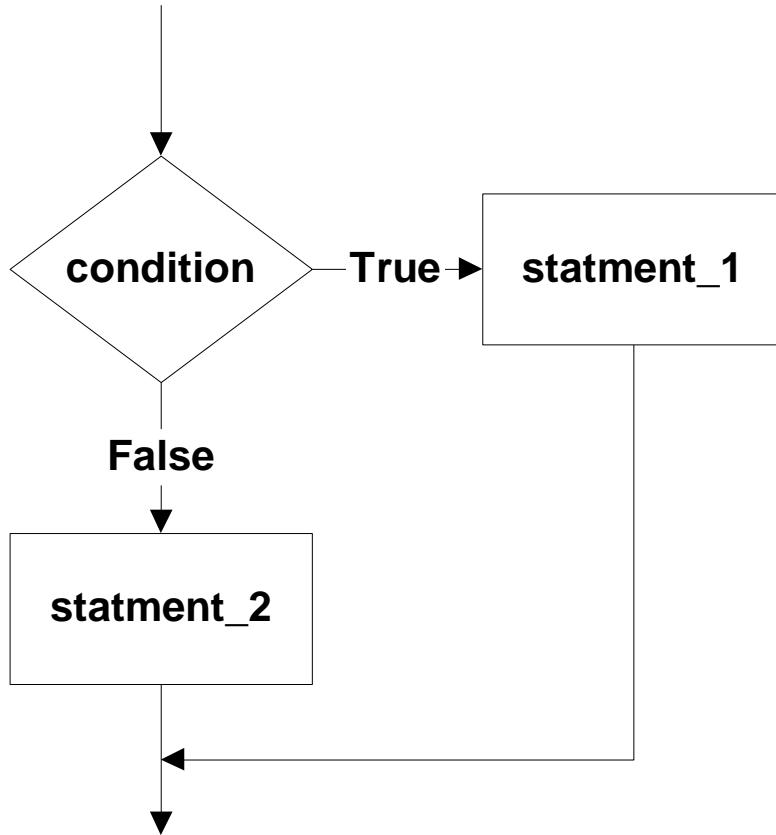
Example 1: Printing a number only if it is a negative



```
if (n<0)
{
    cout << n;
}
```

Translating flowchart to C++ code

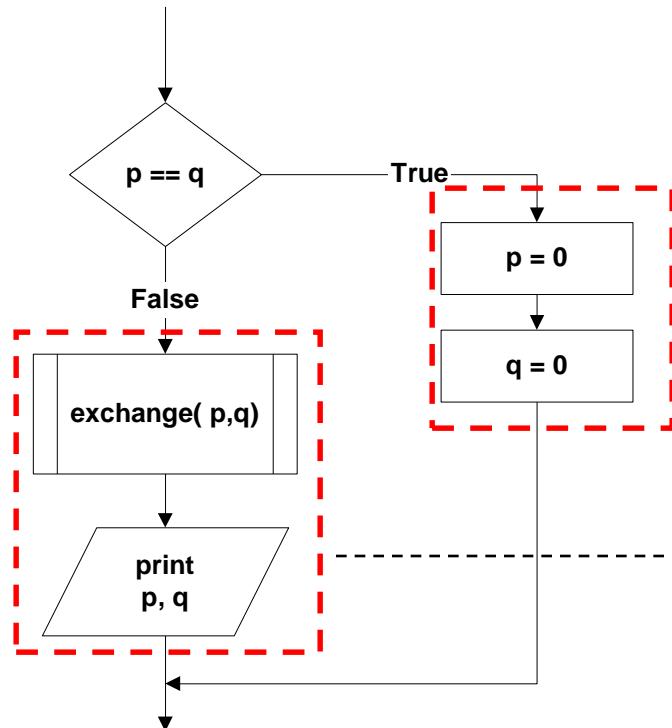
Pattern 2



```
if (condition)
{
    statement_1;
}
else
{
    statement_2;
}
```

Translating flowchart to C++ code

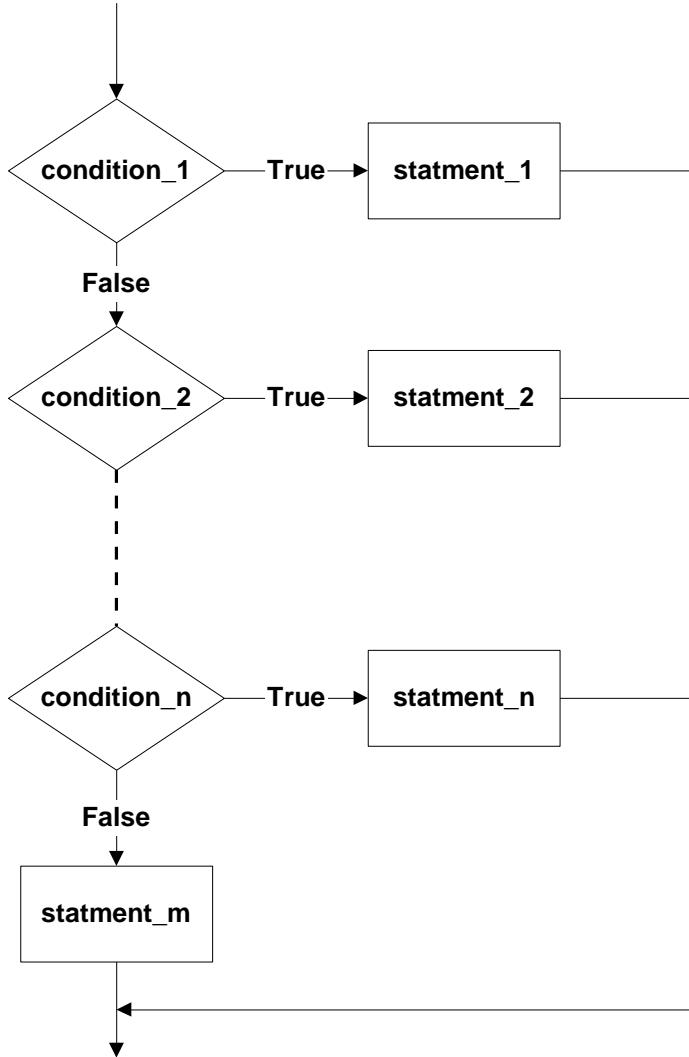
Example 2: If two numbers (p and q) are equivalent reset them to zero, otherwise exchange or swap their value each other and then print the new values.



```

if  (p==q)
{
    p = 0;
    q = 0;
}
else
{
    → exchange (&p , &q) ;
    cout << p << q;
}
  
```

Pattern 3



```

if (condition_1)
{
  statement_1;
}

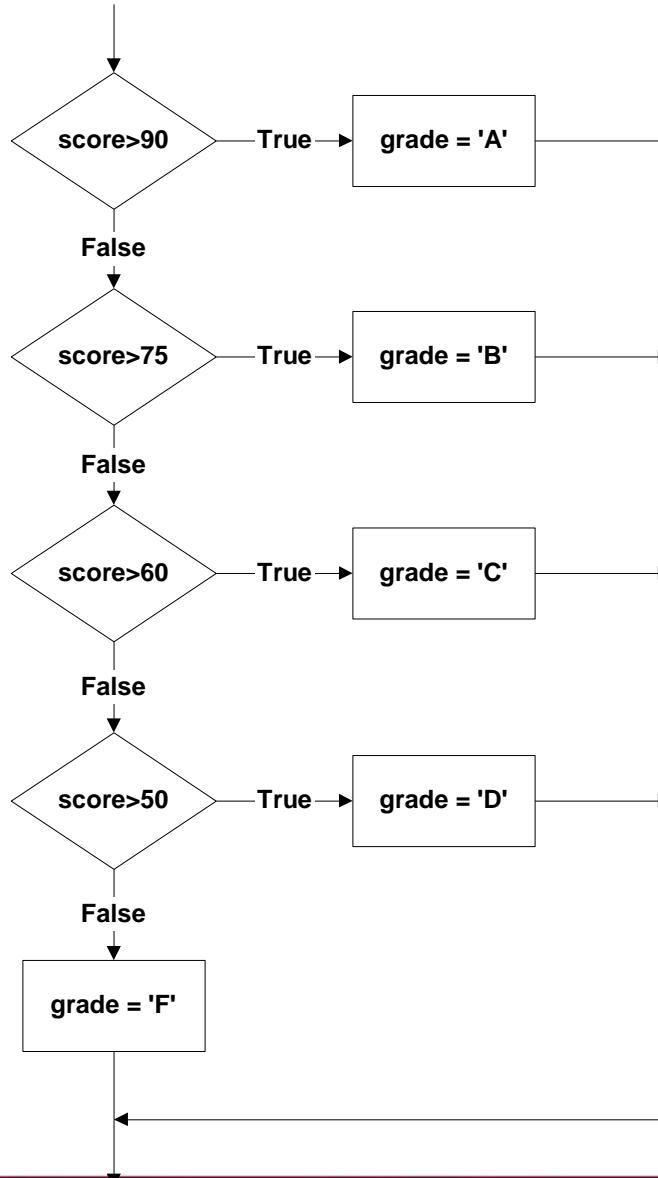
else if (condition_2)
{
  statement_2;
}

else if (condition_n)
{
  statement_n;
}

else
{
  statement_m;
}
  
```

Translating flowchart to C++ code

Example 3: Identifying the grade of a score



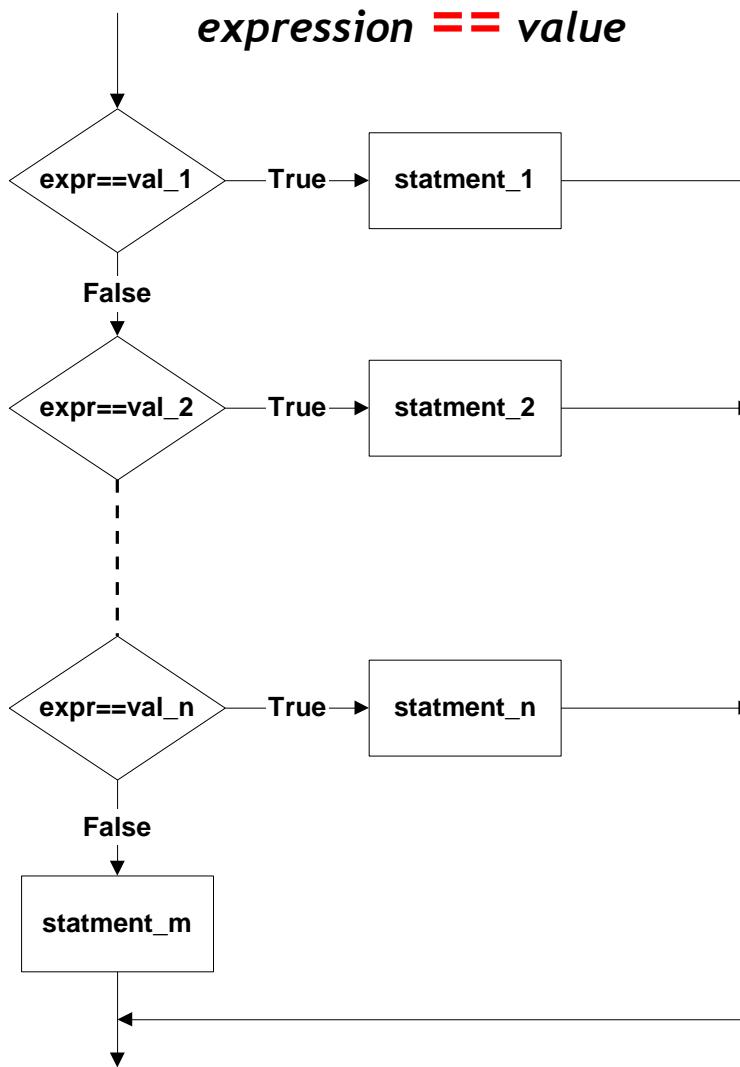
```

if (score > 90)
{
    grade = 'A';
}
else if (score > 75)
{
    grade = 'B';
}
else if (score > 60)
{
    grade = 'C';
}
else if (score > 50)
{
    grade = 'D';
}
else
{
    grade = 'F';
}
  
```

Translating flowchart to C++ code

Pattern 4

- The conditions must be in this form:



```

switch (expr)
{
    case val_1 : statement_1;
                  break;

    case val_2 : statement_2;
                  break;

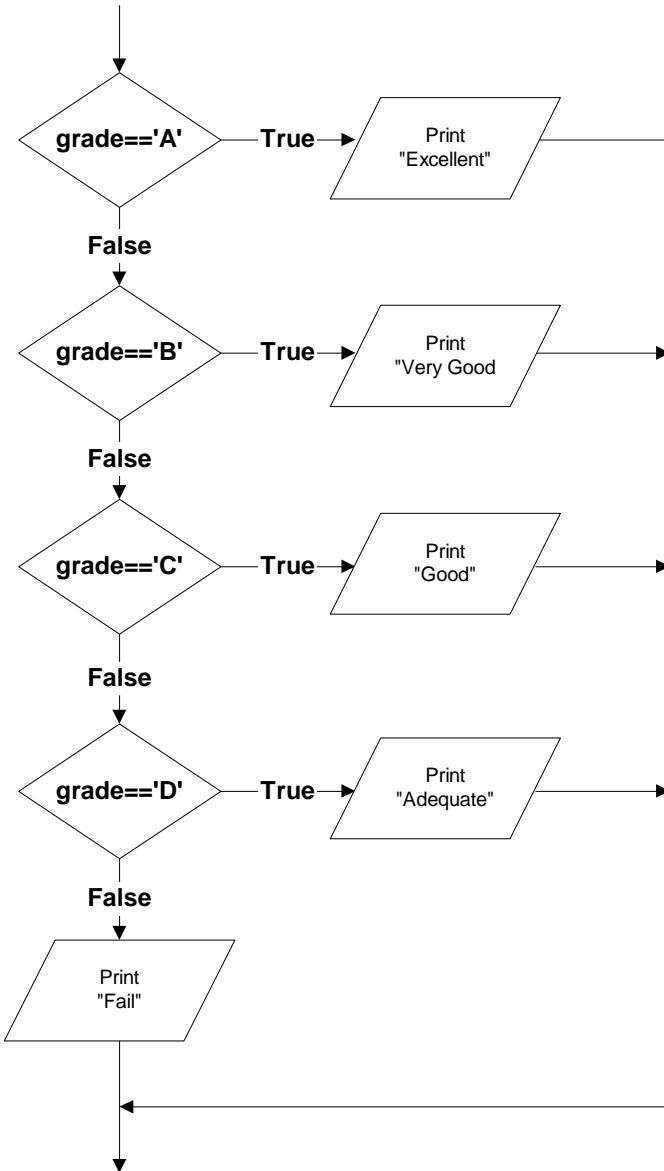
    |
    |

    case val_n : statement_n;
                  break;

    default:     statement_m;
                  break;
}
  
```

Translating flowchart to C++ code

Example 4: Printing the description of a grade.



```

switch (grade)
{
    case 'A' : cout << "Excellent!";
                break;

    case 'B' : cout << "Very good!";
                break;

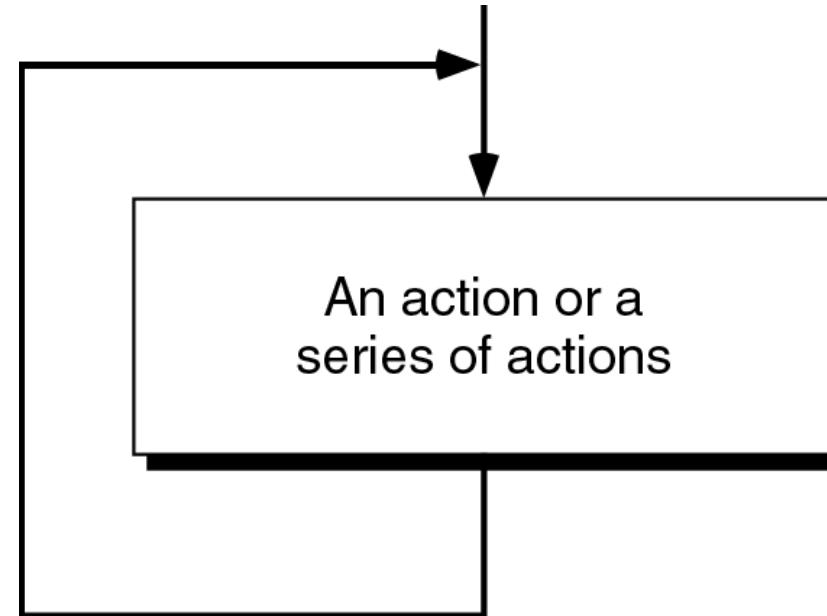
    case 'C' : cout << "Good";
                break;

    case 'D' : cout << "Adequate";
                break;

    default   : cout << "Fail";
                break;
}
  
```

Loop / Repetition

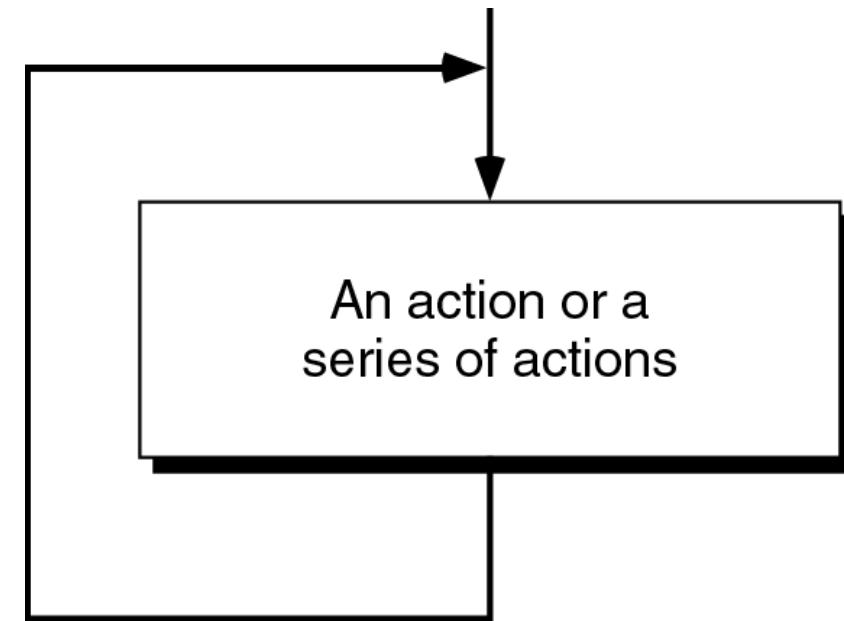
- The main idea of a loop is to **repeat an action or a series of actions.**



The concept of a loop

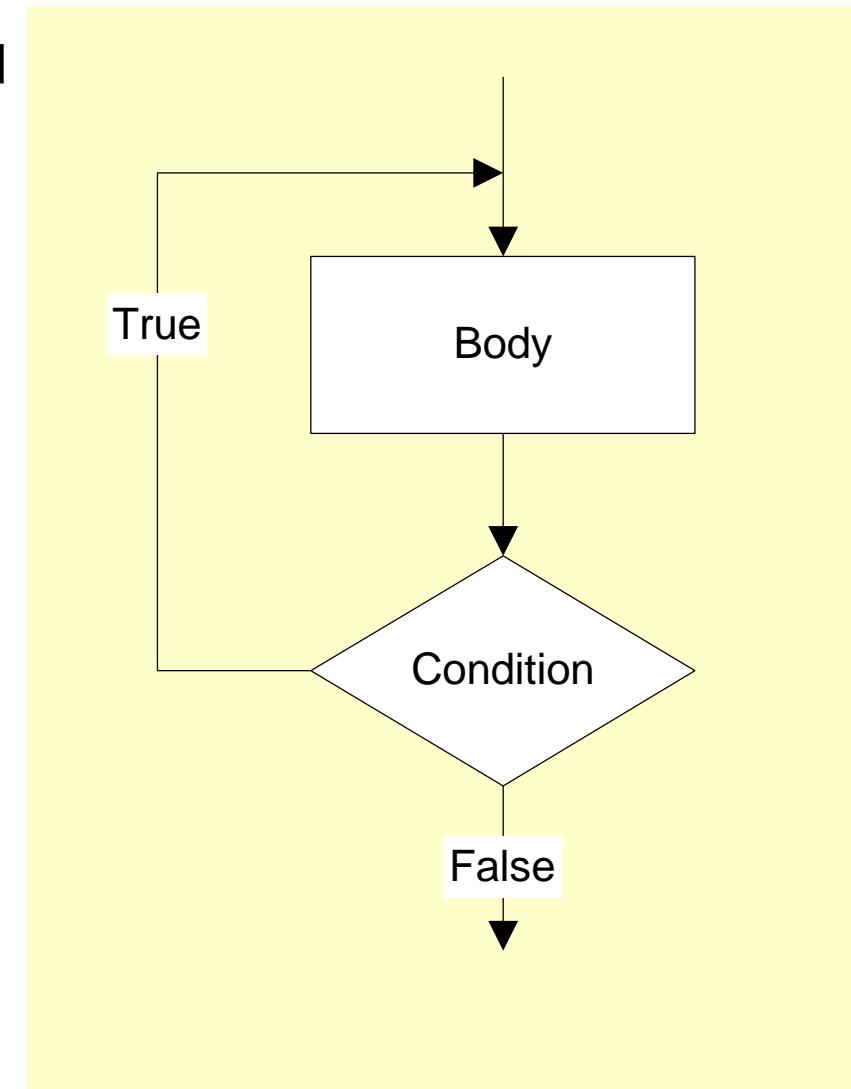
Loops

- But, when to stop looping?
- In the following flowchart, the action is executed over and over again. It never stops - This is called an **infinite loop**
- Solution - put a **condition** to tell the loop either continue looping or stop.



Loops

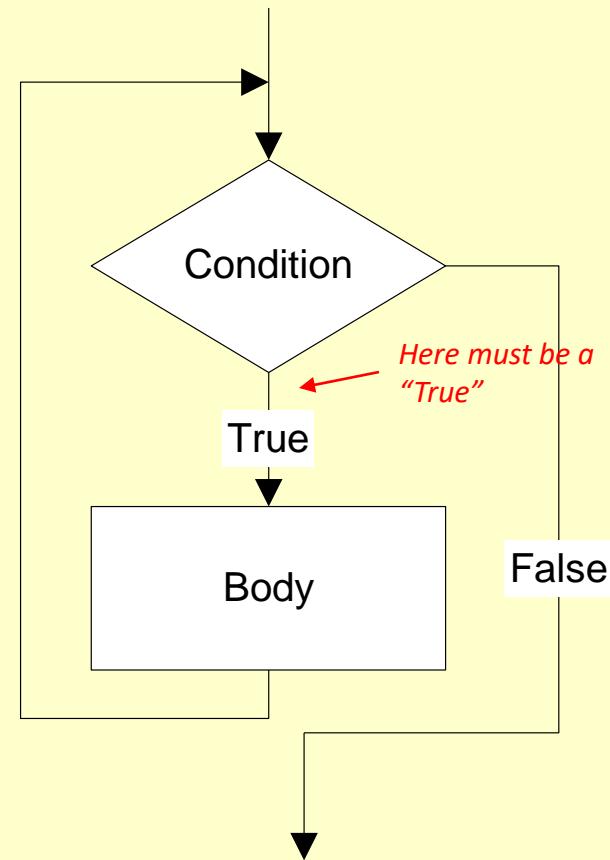
- A loop has two parts - **body** and **condition**
- **Body** - a statement or a block of statements that will be repeated.
- **Condition** - is used to control the iteration - either to continue or stop iterating.



Types of loop

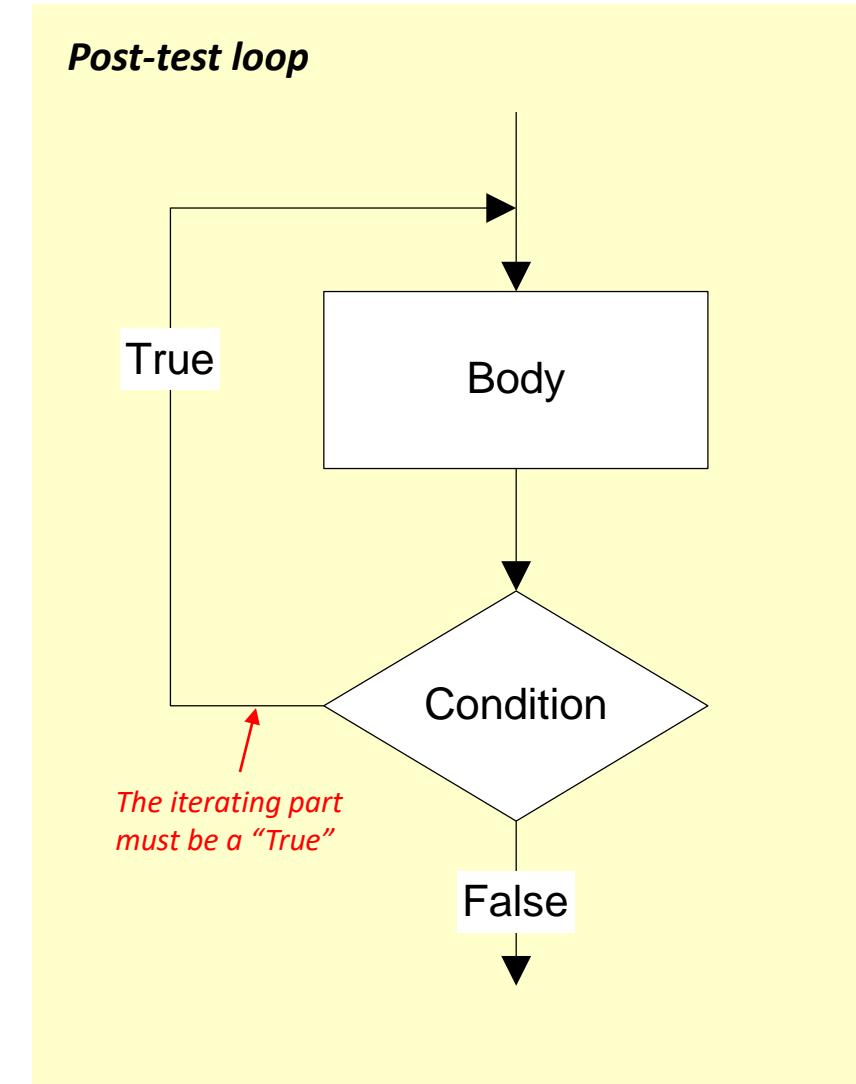
- Two forms of loop - **pretest** loop and **post-test** loop.
- Pretest loop
 - the **condition is tested first**, before we start executing the body.
 - The body is executed if the condition is true.
 - After executing the body, the loop repeats

Pretest loop



Types of loop

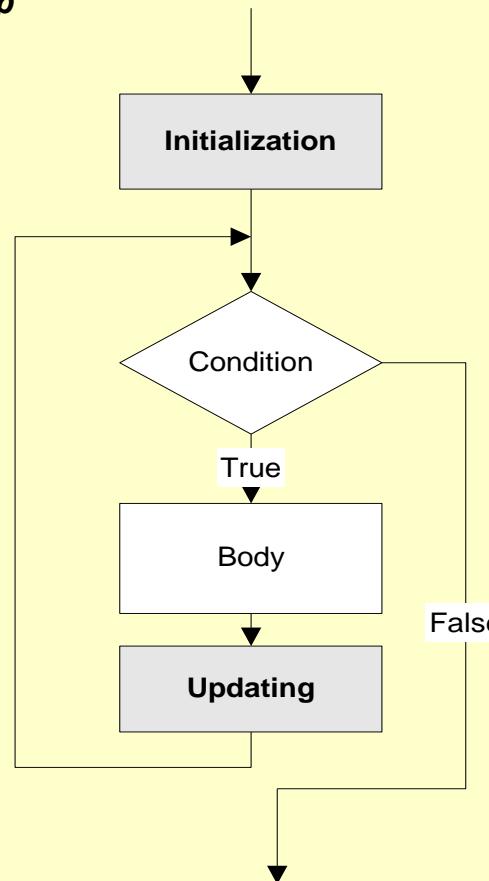
- Post-test loop
 - the condition is tested later, after executing the body.
 - If the condition is true, the loop repeats, otherwise it terminates.
 - The body is always executed at least once.



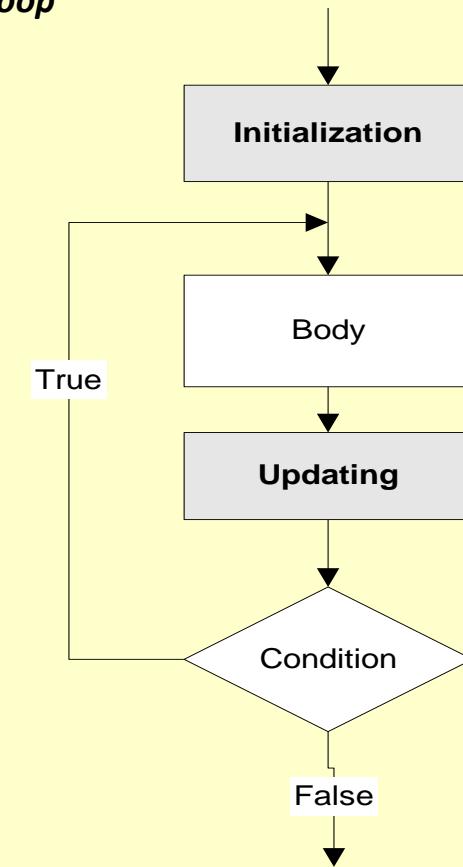
Parts of a loop

- Beside the body and condition, a loop may have two other parts - **Initialization** and **Updating**

Pretest loop



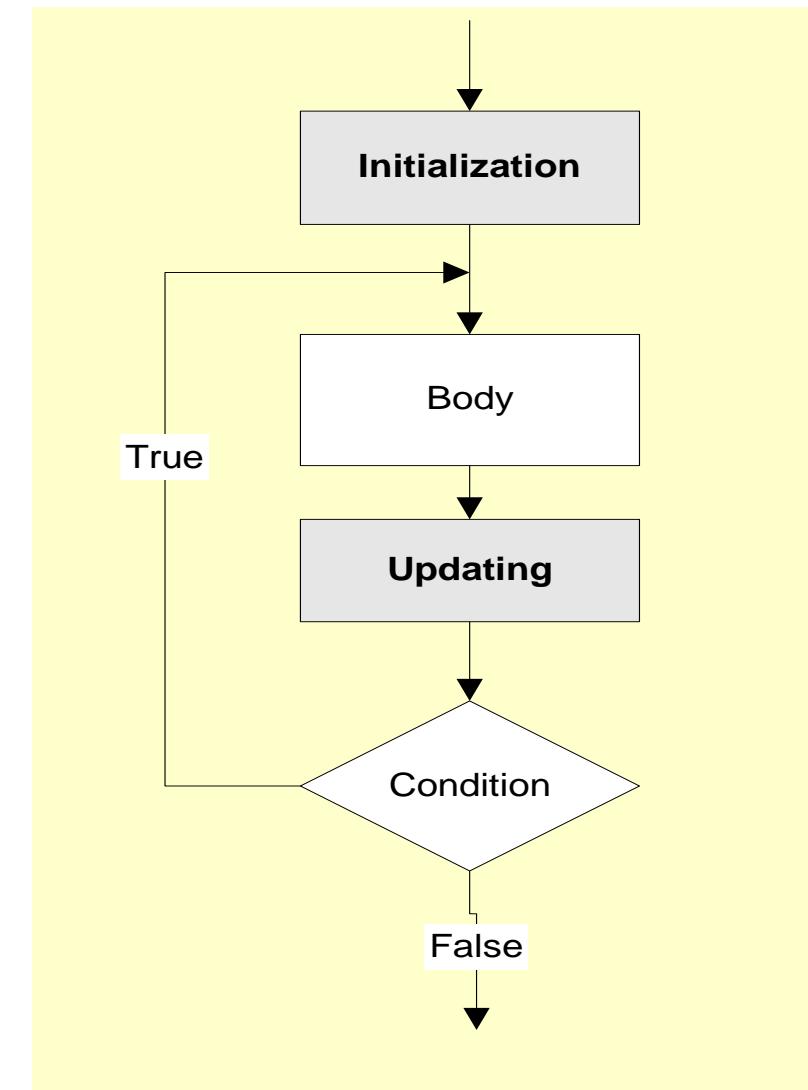
Post-test loop



Parts of a loop

- **Initialization**
 - is used to prepare a loop before it can start -usually, here we **initialize the condition**
 - The initialization must be written outside of the loop - before the first execution of the body.

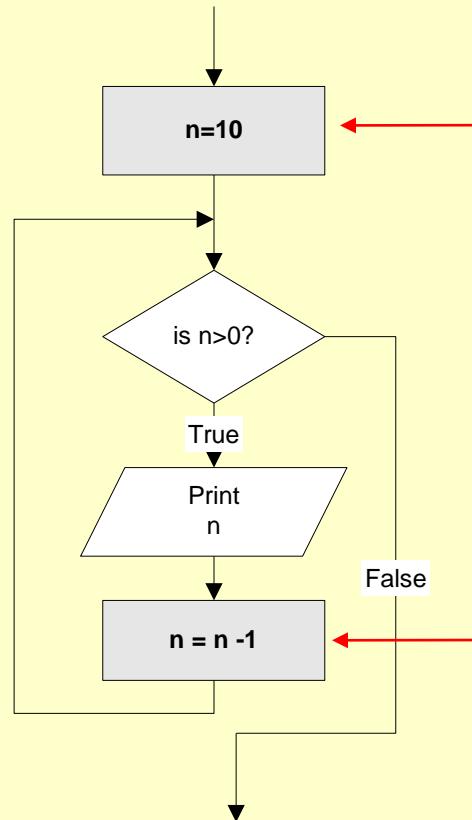
- **Updating**
 - is used to **update the condition**
 - If the condition is not updated, it always true => the loop always repeats
 - an **infinite loop**
 - The updating part is written inside the loop - it is actually a part of the body.



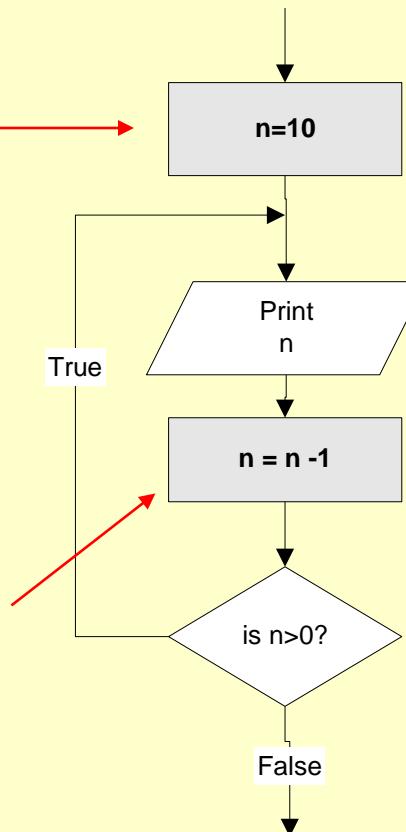
Parts of a loop

Example: These flowcharts print numbers 10 down to 1

Pretest loop



Post-test loop

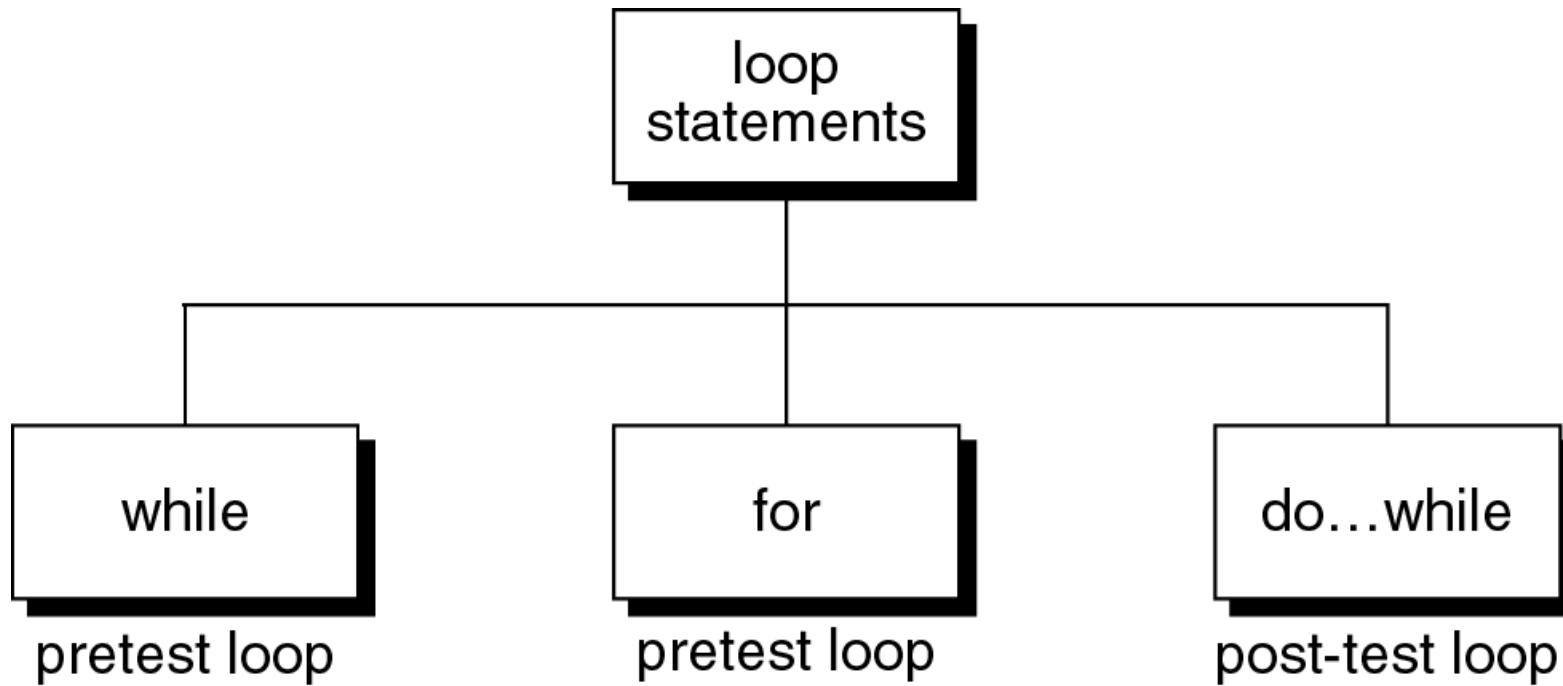


Initialize n before start the loop

Every time the loop repeats, n is updated

Loop statements

- C++ provides three loop statements:

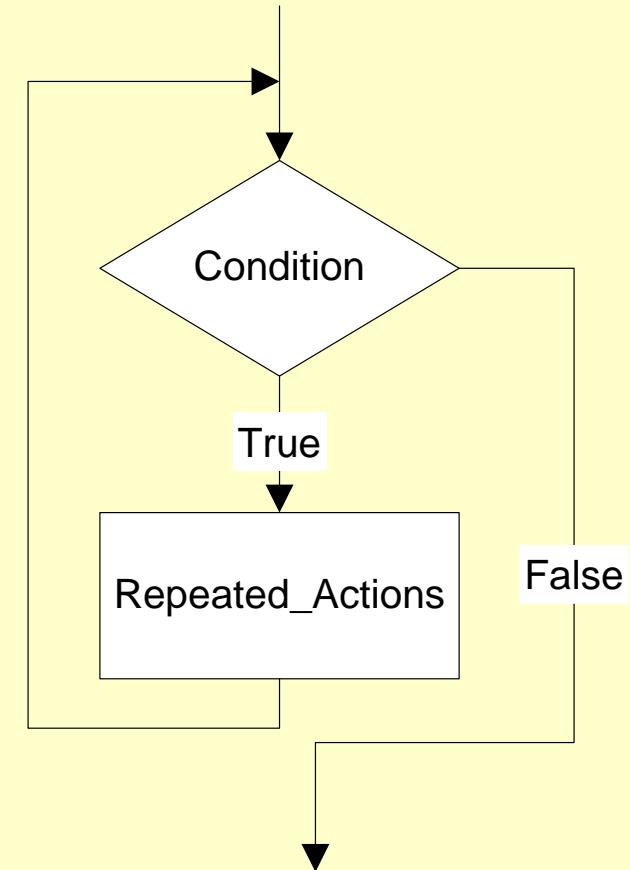


C++ loop constructs

while statement

```
while (Condition)
{
    Repeated_Actions;
}
```

while flowchart

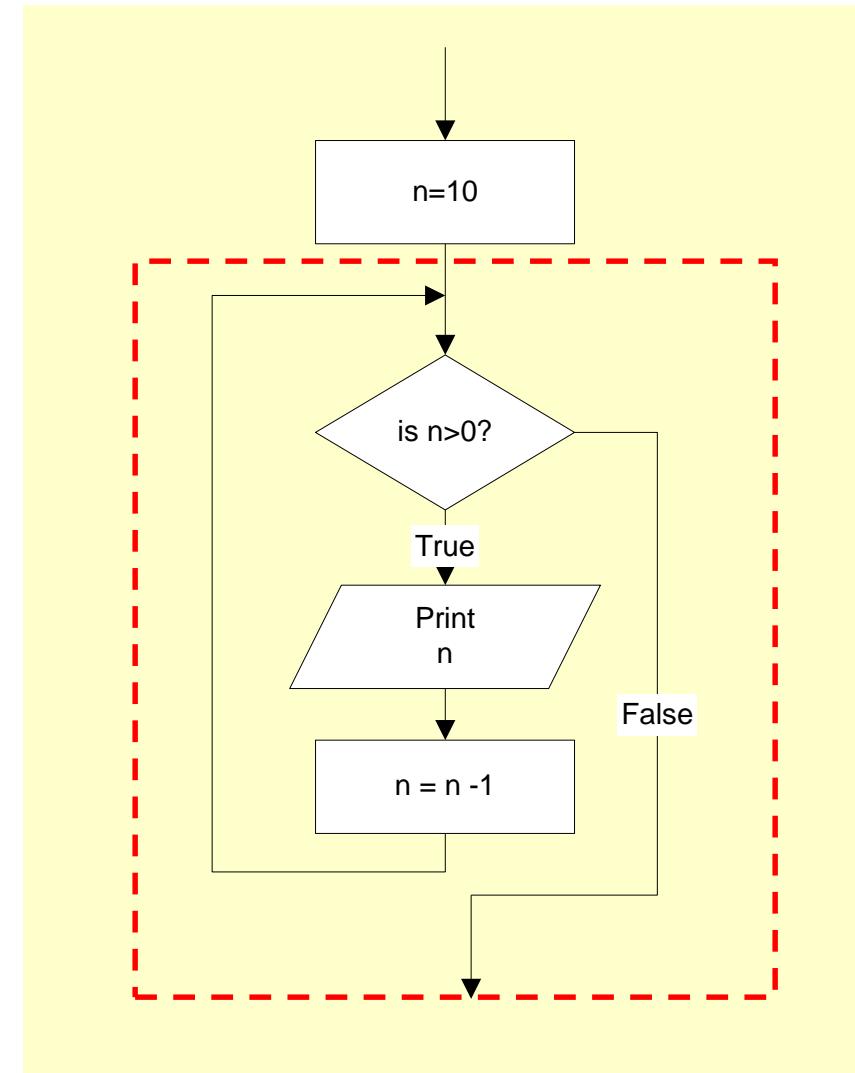


while statement

Example: This while statement prints numbers 10 down to 1

Note that, the first line ($n=10$) is actually not a part of the loop statement.

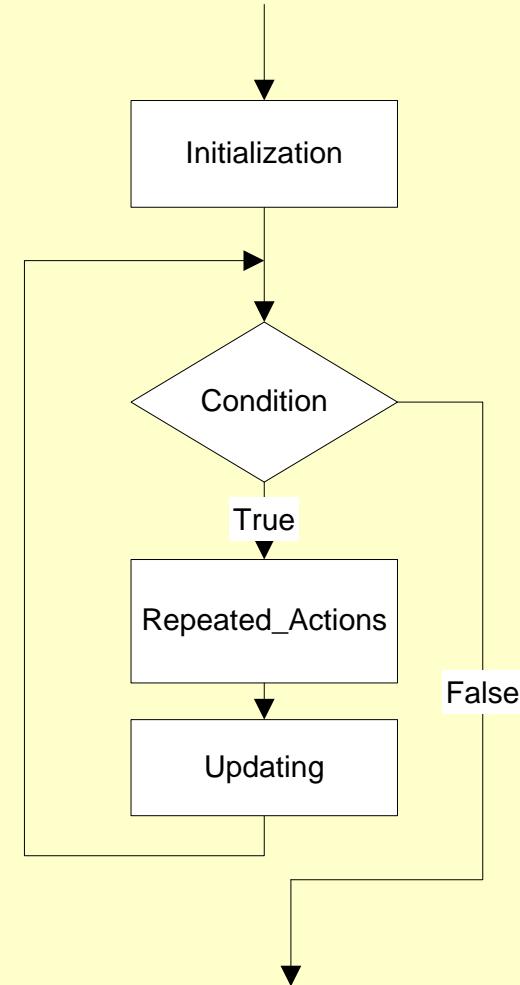
```
n=10;  
  
while (n>0)  
{  
    cout << n <<" ";  
    n=n-1;  
}
```



for statement

```
for (Initialization; Condition; Updating)  
{  
    Repeated_Actions;  
}
```

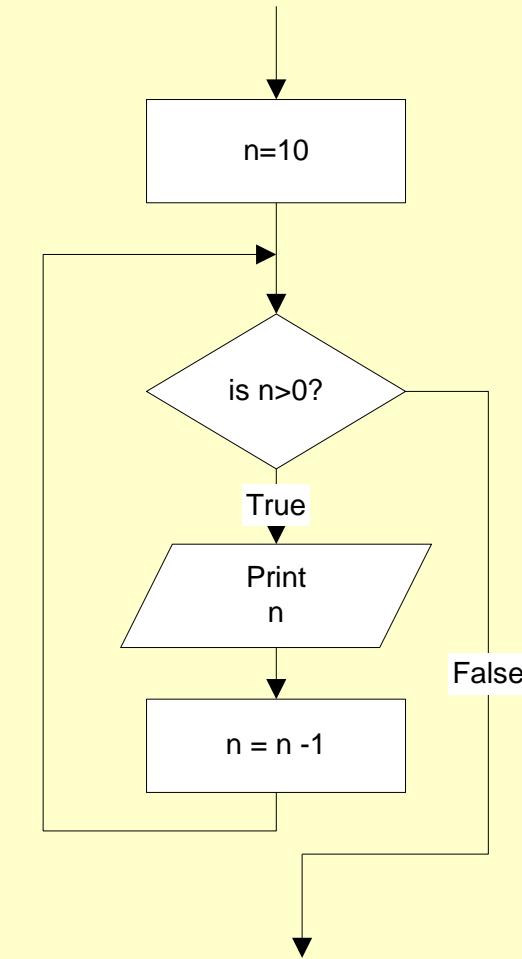
for flowchart



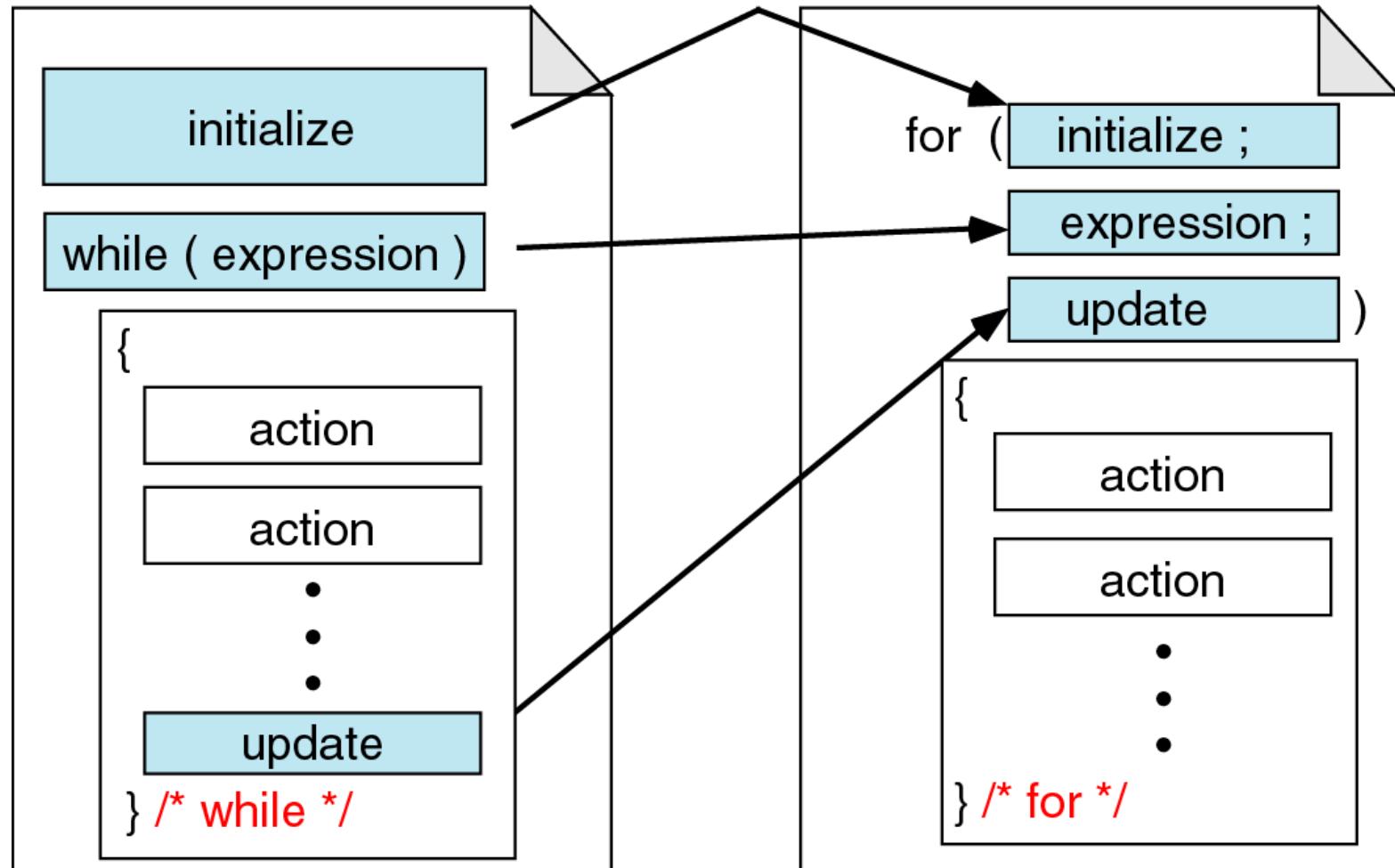
for statement

Example: This for statement prints numbers 10 down to 1

```
for (n=10; n>0; n=n-1)
{
    cout << n << " ";
}
```



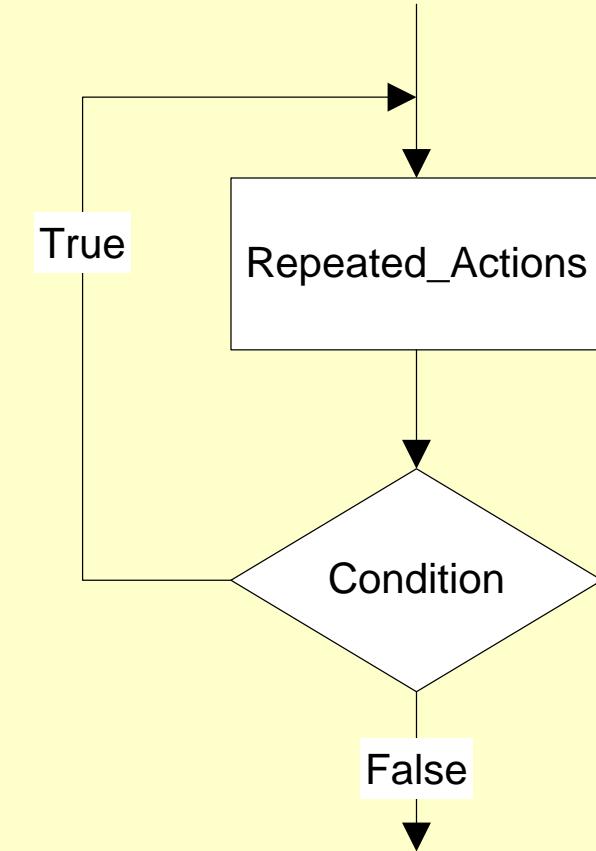
for vs. while statements



Comparing `for` and `while` loops

do...while statement

```
do
{
    Repeated_Actions;
} while (Condition);
```

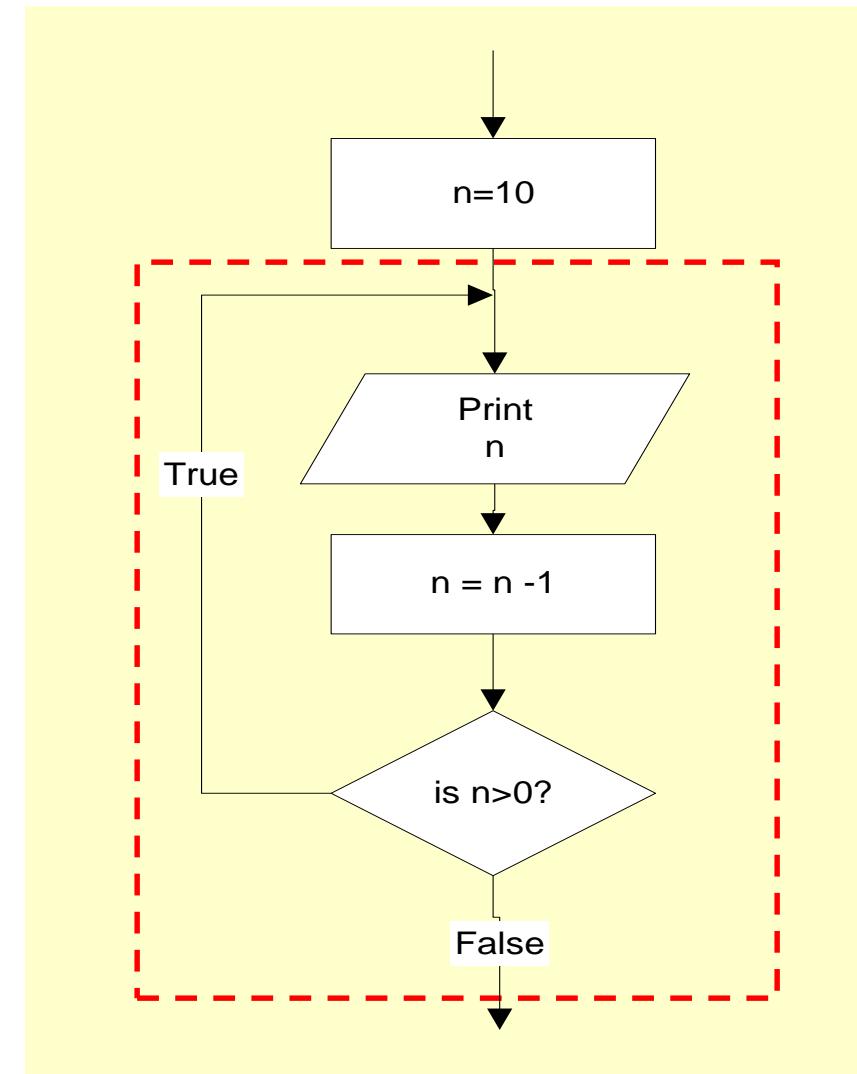


do...while statement

Example: This do...while statement prints numbers 10 down to 1

Note that, the first line ($n=10$) is actually not a part of the loop statement.

```
n=10;  
do  
{  
    cout << n << " ";  
    n=n-1;  
} while (n>0);
```



Loop statements

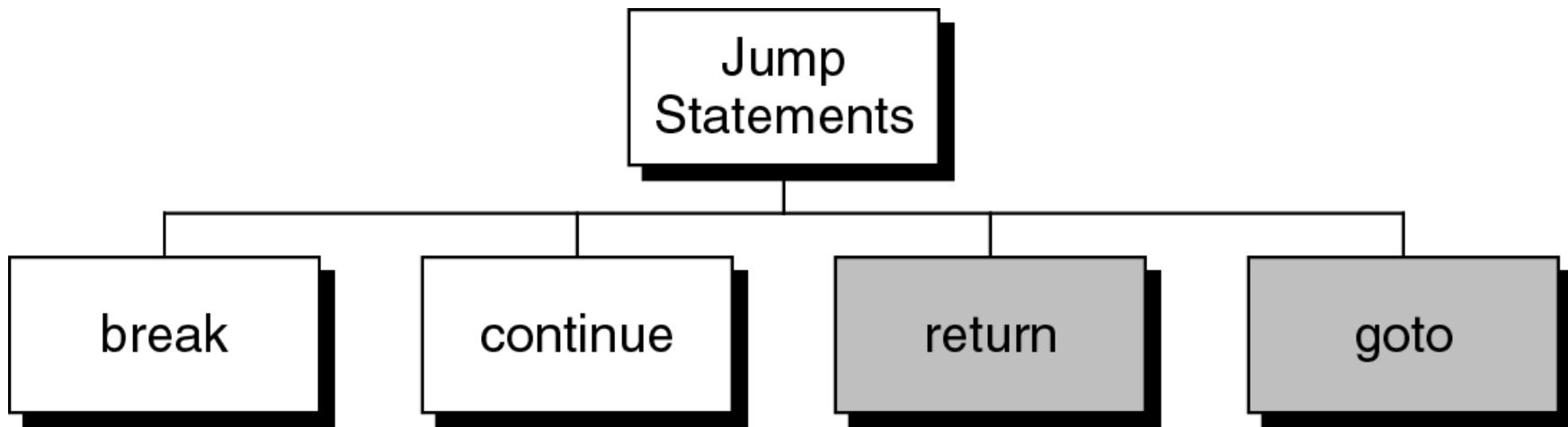
- If the body part has only **one statement**, then the bracket symbols, **{ }** may be omitted.
- Example: These two `for` statements are equivalent.

```
for (n=10; n>0; n=n-1)
{
    cout << n;
}
```

```
for (n=10; n>0; n=n-1)
    cout << n;
```

Jump statements

- You have learned that, the repetition of a loop is controlled by the loop condition.
- C++ provides another way to control the loop, by using **jump statements**.
- There are four jump statements:



Breaking Out of a Loop

- Can use **break** to terminate execution of a loop
- Use sparingly if at all – makes code harder to understand
- When used in an inner loop, terminates that loop only and returns to the outer loop

break statement

- It causes a loop to **terminate**

Example:

```
for (n=10; n>0; n=n-1)
{
    if (n<8) break;
    cout << n << " ";
}
```

break statement

```
while (condition)
{
    ...
    for ( ...; ...; ... )
    {
        ...
        if (otherCondition)
            break;
        ...
    } /* for */
    ...
} /* while */
```

The break statement takes you out of the inner loop (the *for* loop). The *while* loop is still active.

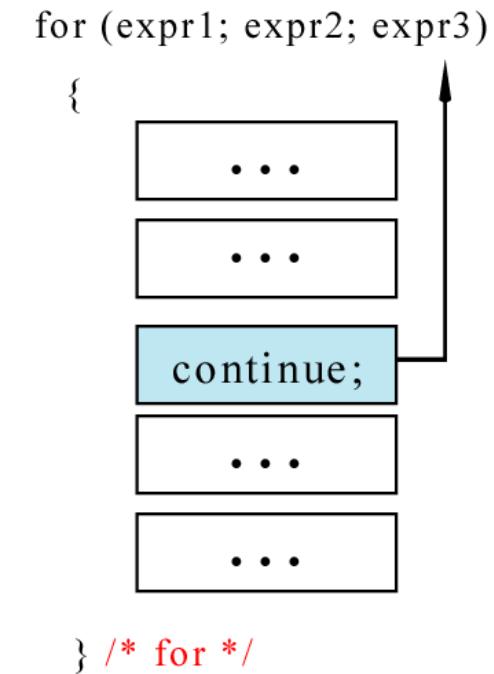
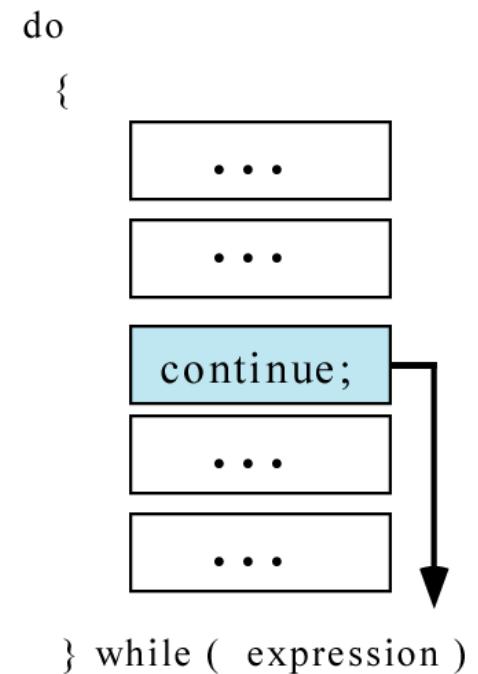
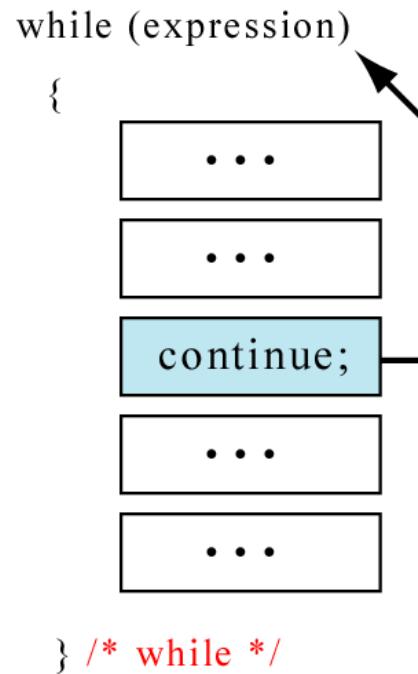
***break* an inner loop**

The **continue** Statement

- Can use **continue** to go to end of loop and prepare for next repetition
 - **while** and **do-while** loops go to test and repeat the loop if test condition is true
 - **for** loop goes to update step, then tests, and repeats loop if test condition is true
- Use sparingly – like **break**, can make program logic hard to follow

continue statement

- In while and do...while loops, the continue statement transfers the control to the loop condition.
- In for loop, the continue statement transfers the control to the updating part.



The **continue** statement

continue statement

Example:

```
for (n=10; n>0; n=n-1)
{
    if (n%2==1) continue;
    cout << n <<" ";
}
```

continue statement

Example:

```
n = 10;  
while (n>0)  
{  
    cout << n << " ";  
    if (n%2==1) continue;  
    n = n -1;  
}
```

return statement

- You will learn this statement in Chapter 4 - Function.
- It causes a **function to terminate**.

Example:

```
void print_numbers()
{ int n=10;
  int i;

  while (n>0)
  {
    for (i=n;i>0; i--)
    {
      if (i%2==1) continue;

      if (i%4==0) break;

      if (n==6) return;

      cout <<i <<" ";
    }
    cout << endl;
    n=n-1;
  }
}
```

The **continue** statement transfers control to the updating part (*i--*)

The **break** statement terminates the **for loop**.

The **return** statement terminates the **function** and returns to the caller.

return statement

- When to use `return`?
- *Example:* the following functions are equivalent

```
float calc_point(char grade)
{
    float result;

    if (grade=='A') result = 4.0;
    else if (grade=='B') result = 3.0;
    else if (grade=='C') result = 2.5;
    else if (grade=='D') result = 2.0;
    else result = 0.0;

    return result;
}
```

```
float calc_point(char grade)
{
    if (grade=='A') return 4.0;
    if (grade=='B') return 3.0;
    if (grade=='C') return 2.5;
    if (grade=='D') return 2.0;
    return 0.0;
}
```

The *else* part of each *if* statement may be omitted. It has never been reached.



return statement

```
float calc_point3(char grade)
{
    float result;

    switch (grade)
    {
        case 'A': result = 4.0;
                    break;

        case 'B': result = 3.0;
                    break;

        case 'C': result = 2.5;
                    break;

        case 'D': result = 2.0;
                    break;

        default:   result = 0.0;
    }

    return result;
}
```

```
float calc_point4(char grade)
{
    switch (grade)
    {
        case 'A': return 4.0;

        case 'B': return 3.0;

        case 'C': return 2.5;

        case 'D': return 2.0;
    }
    return 0.0;
}
```

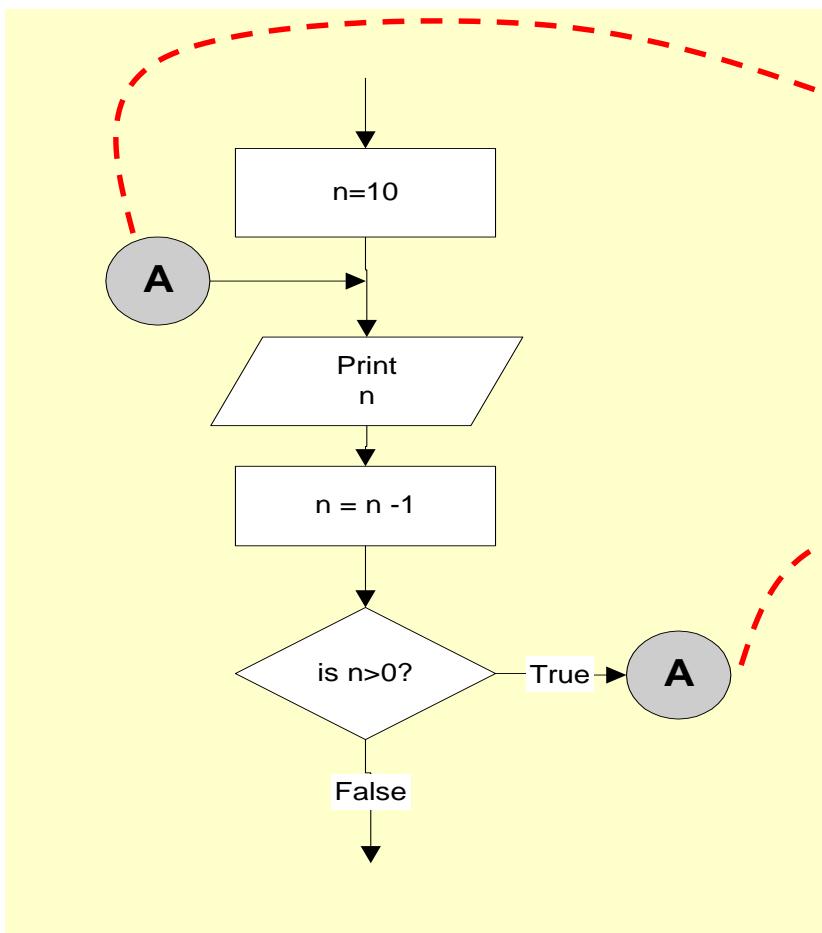
The *break* statement of each *case* may be omitted. It has never been reached.



goto statement

- It is used to translate **connector symbols** - jump to another part inside a program.
- But, it is not recommended to use - **it may cause unstructured programs.**

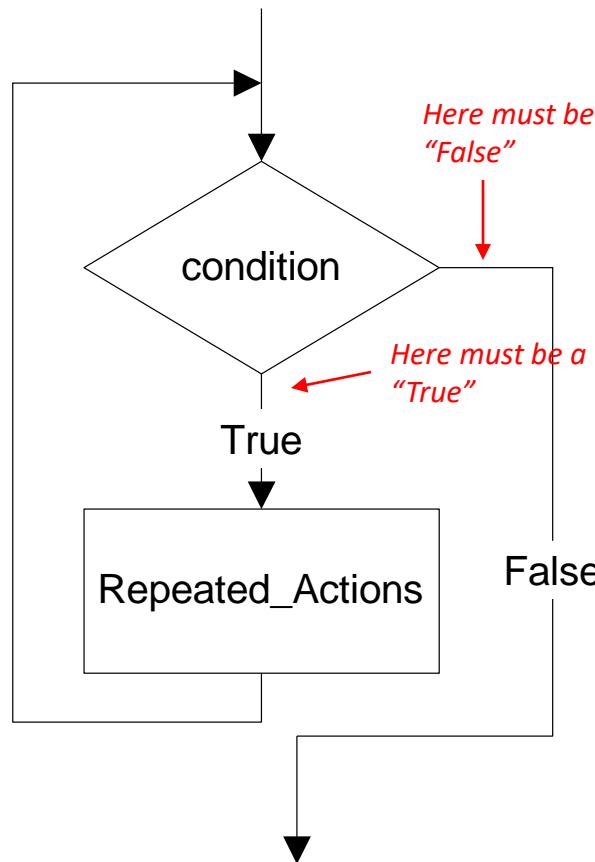
Example:



```
n=10;  
  
A:  
    cout <<n <<" "  
    n = n -1;  
  
    if (n>0) goto A;
```

Translating flowchart to C++ code

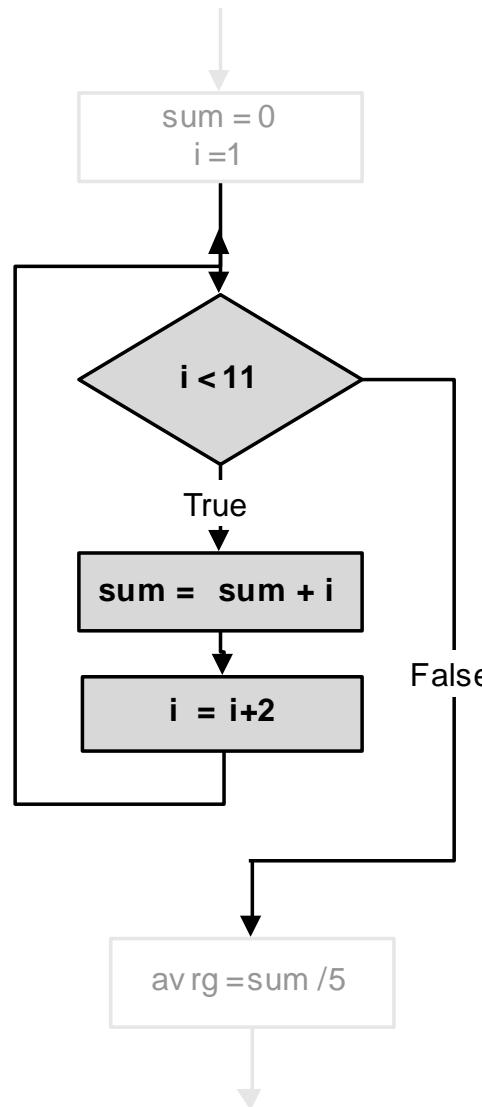
Pattern 1



```
while (condition)
{
    Repeated_Actions;
}
```

Translating flowchart to C++ code

Example: Calculate the average of odd numbers 1 to 9

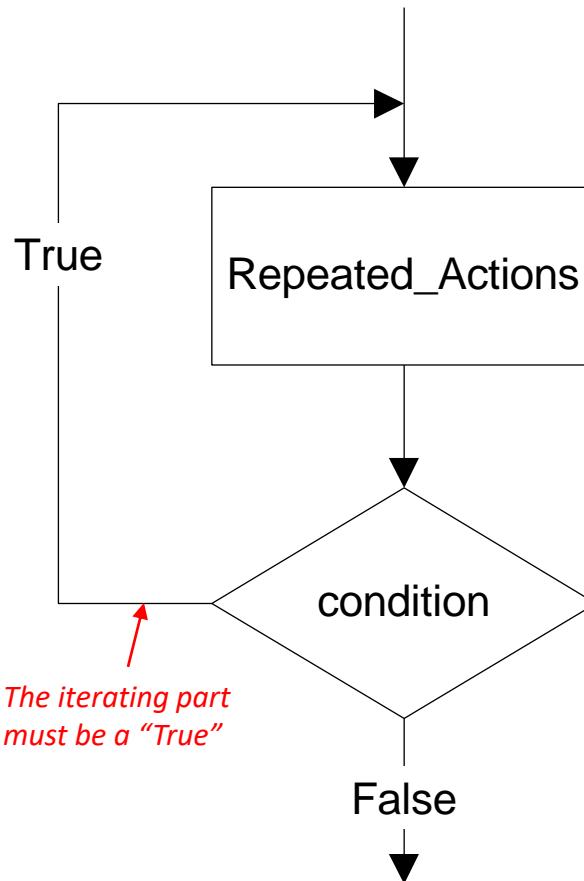


```

sum = 0;
i=1;
while (i<11)
{
    sum = sum + i;
    i = i + 2;
}
avrg = sum/5.0;
  
```

Translating flowchart to C++ code

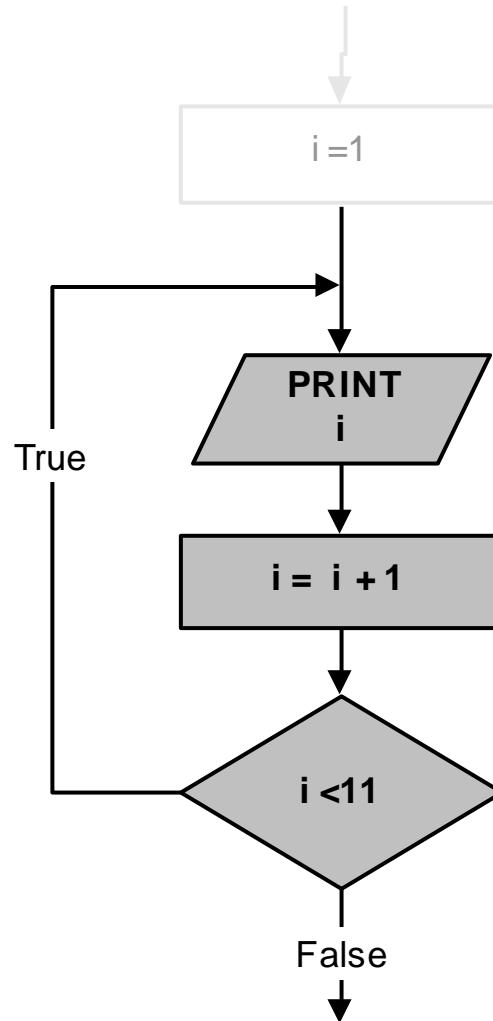
Pattern 2



```
do
{
    Repeated Actions;
} while(condition);
```

Translating flowchart to C++ code

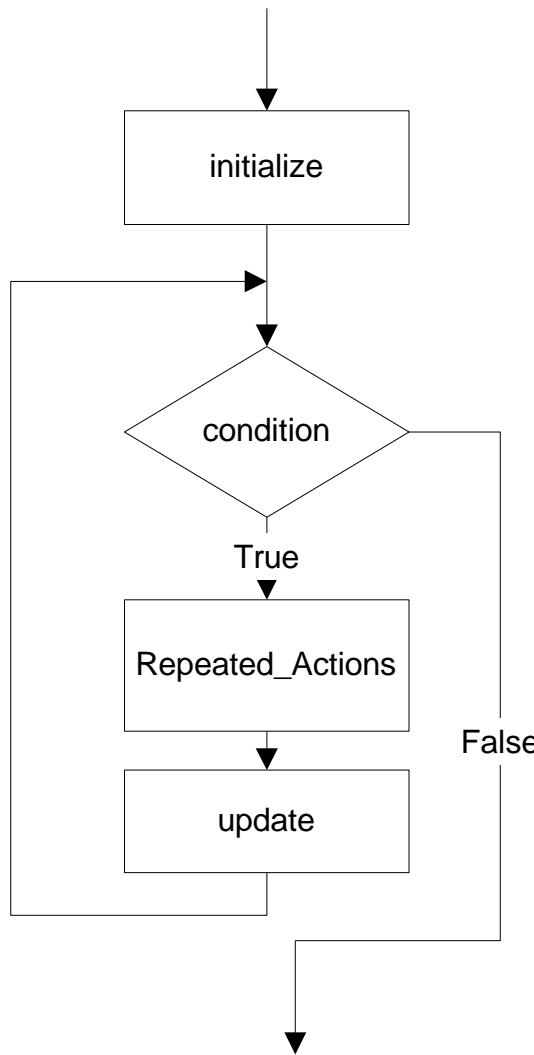
Example: Prints numbers 1 to 10



```
i=1;  
do  
{  
    cout <<i <<endl;  
    i = i + 1;  
} while (i<11);
```

Translating flowchart to C++ code

Pattern 3



```

for (initialize; condition; update)
{
  Repeated Actions;
}
  
```

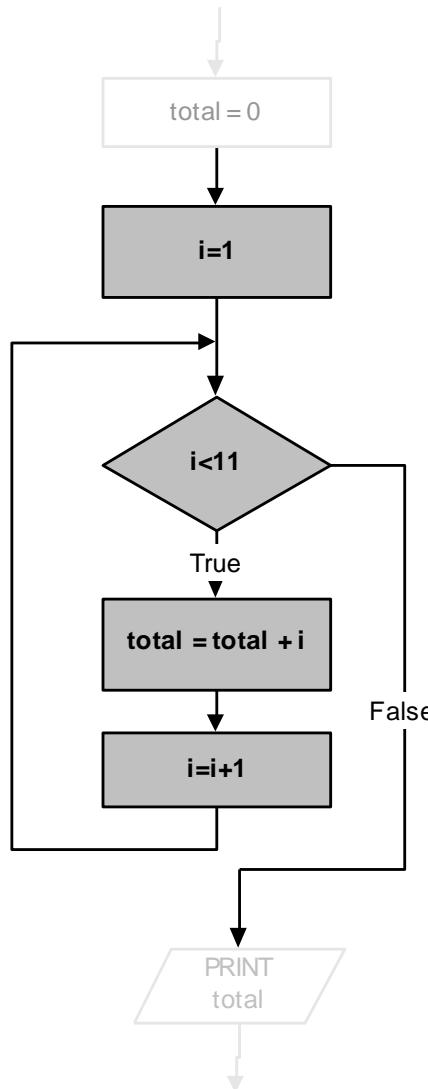
or

```

initialize;
while (condition)
{
  Repeated Actions;
  update;
}
  
```

Translating flowchart to C++ code

Example: Print the total of numbers 1 to 10



```

total = 0;
for (i=1; i<11; i++)
{
  total = total + i;
}
cout <<total;
  
```

or

```

total = 0;
i=1;
while (i<11)
{
  total = total + i;
  i++;
}
cout <<total;
  
```

Deciding Which Loop to Use

- **while**: pretest loop (loop body may not be executed at all)
- **do-while**: post test loop (loop body will always be executed at least once)
- **for**: pretest loop (loop body may not be executed at all); has initialization and update code; is useful with counters or if precise number of repetitions is known

Nested Loops

- A **nested loop** is a loop inside the body of another loop
- Example:

```
for (row=1; row<=3; row++) outer loop
{
    for (col=1; col<=3; col++) inner loop
    {
        cout << row * col << endl;
    }
}
```

Notes on Nested Loops

- Inner loop goes through all its repetitions for each repetition of outer loop
- Inner loop repetitions complete sooner than outer loop
- Total number of repetitions for inner loop is product of number of repetitions of the two loops. In previous example, inner loop repeats 9 times

In-Class Exercise

- How many times the outer loop is executed? How many times the inner loop is executed? What is the output?

```
#include <iostream>
using namespace std;
int main()
{
    int x, y;
    for(x=1;x<=8;x+=2)
        for(y=x;y<=10;y+=3)
            cout<<"\nx = " <<x << "    y = "<<y;
    return 0;
}
```