Conditional Execution

Ajit Rajwade

Refer: Chapter 6 of the book by Abhiram Ranade

The need for conditional statements

- Suppose you want to write a program which takes as input your percentage score in a certain course, and determines whether or not you will secure an AA.
- Consider the rule that you get an AA if and only if you scored more than 90 percent.
- The statement `if (condition) consequent;` helps us to write programs like these.

Example: program with `if` statement

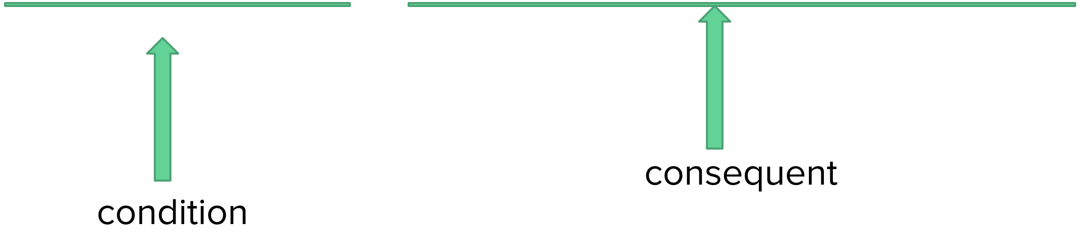
```
main_program{

float score;

cout << “enter your score: “; cin >> score;

if (score > 90.0) cout << “You got an AA”;

}
```



condition

consequent

The condition needs to be an expression that evaluates to either true or false. If the condition is true, the consequent is executed, otherwise it is ignored.

Conditions of different types

- The condition involves **comparison or logical operators** such as `>`, `<` , `>=`, `<=`, `==`, `!=`.
- The first four above have their usual meanings.
- The `==` operator checks whether the expressions on both sides evaluate to the same, whereas `!=` checks whether they are not equal.
- The `==` operator should not be confused with the assignment operator `=`.
- The conditions can also be compound. For example, if you scored more than or equal to 85 percent **and** less than or equal to 90, then you get an AB, i.e. `if (score >= 85 && score <= 90) cout << “You got an AB”;`
- The “and” part above is represented by the operator `&&` called a **conjunction**. There is also an operator `&` but it means something else.
- For the condition in the `if` statement to evaluate to true, **both** `score >= 85` **and** `score <= 90` **must** evaluate to true.

Conditions of different types

- Imagine you had a course where you could get an AA by scoring more than 90 percent **or** scoring 100 percent on a semester-long course project.
- The or part is expressed as follows: `if  (score > 90 || project_score == 100) cout << “You got an AA”;`
- The `||` operator, called a **disjunction**, evaluates to true if either `score > 90` **or** `project_score == 100` **or both** evaluate to true.
- There also exists an operator `|` but it works very differently from `||`.
- The operators `&&` and `||` can be combined with each other and create a compound condition from many constituent conditions:

```
C1 || C2 || C3 || … || CN

C1 && C2 && C3 && … && CN

C1 || C2 && C3 || C4
```

Conditions of different types

- There is one more conditional operator `! condition` which just **negates** the `condition` that follows it.
- Consider `if (score > 90.0) cout << “You got an AA”;`
- Or equivalently consider: `if (!(score <= 90.0)) cout << “You got an AA”;`

Block conditionals

- The consequent need not be a single statement. It can be an entire block.
- Example:

```
int number_AAs = 0;
if (score > 90.0) {
cout << “You got an AA”;
number_AAs++;
}
```

Block conditionals

- `if` statements can be put inside `repeat` loops.
- Example:

```
int number_AAs = 0;

repeat(number_students_class){

    cout << “enter your score:”; cin >> score;

    if (score > 90.0) {

    cout << “You got an AA”;

    number_AAs++;

    } // close if

} // close repeat
```

Watch out!

- Replacing == by = is a common programming mistake which can lead to **serious** logical errors.
- Example:

```
if (p == 2) cout << “p is an even prime number”;

if (p = 2) cout << “p is an even prime number”;
```

- The latter statement will **assign the value 2 to** **`p`** **erasing its earlier value** and then print “`p is an even prime number`”.
- The former statement **checks whether p is equal to 2** and **if true**, prints out “`p is an even prime number`”. Either way, the value of `p` is left intact.

Consequents and Alternatives

- These are statements of the form:

```
if condition (consequent);

else alternative;

OR

if  (condition1) consequent1;

else if (condition2) consequent2;

.

.

else if (conditionN) consequentN;

else alternative;
```

- Each (or some subset) of the consequents and the alternative can be a block of statements instead of being a single statement.

Consequents and Alternatives

The portions of the conditions in red are not really required. It can be removed, but has been retained for illustration only.

```
main_program{

float score;

cout << “enter your score: “; cin >> score;

if (score > 90.0) cout << “You got an AA”;

else if (score  > 85 && score <= 90) cout << “You got an AB”;

else if (score > 80 && score <= 85) cout << “You got a BB”;

else cout << “You scored less than a BB”;

}
```

Nested `if` statements

- The `if` statement and `if else` statements can be nested inside each other.
- For example:

```
if (condition1)

{

    if (condition 2) consequent 1;

    else alternative2;

}

else alternative1;
```

Watch out!

- The statement `if (a > 0) if (b > 0) c = 5; else c = 6;` can be interpreted in the following two ways:

➔ `if (a > 0) { if (b > 0) c = 5; else c = 6; } // C++ interpets it this way`

➔ `if (a > 0) {if (b > 0) c = 5;} else c = 6; // C++ does not interpret it this way`

- The statement above is an example of correct but confusing code.
- It is best to put brace brackets in the appropriate place to avoid needless confusion such as this!

The `switch` case

- Sometimes a certain variable can take on one from a finite number of **different values**.
- Based on each of these values, a different set of consequent(s) can be executed.
- For example, let us suppose an integer from 1 to 12 represents the different months of the year from January (1) through to December (12) respectively.
- Depending on which month it is, you want to print the number of days in that month.
- You can of course use `if else` statements, but another way to write code for this is the `switch` statement.
- The syntax of the `switch` statement is on the next slide.

The `switch` case: syntax

```
switch (expr1}{

case value1:  block1; break;

case value2: block2; break;

.

.

case valueN: blockN; break;

default: block_default;

}
```

The `expr1` is evaluated. If its value equals `value1`, then the statements in `block1` are executed. When a `break` is encountered, the `switch` loop is exited.

If the `break` statement were missing after `block1`, the program would execute `block2` and possibly other blocks as well as `block_default` until a `break` statement is encountered or if the end of the `switch` case is encountered. This is called a **fall through**.

Likewise if `expr1` evaluated to `value2`, then `block2` and possibly other blocks including `block_default` would be executed until the appearance of the first `break` statement. If `expr1` evaluates to something unequal to all of `value1`, `value2`, ..., `valueN`, then only `block_default` is executed.

The portion consisting of `default` and `block_default` is optional. `block_default` is executed if `expr1` does not match any of the specified values.

All values in the cases must be integers!

`switch` case: example program

```
main_program{
int month; cin >> month;
switch(month){
     case 1: // January
     case 3:
     case 5:
     case 7:
     case 8:
     case 10:
     case 12: cout << “This month has 31 days”; break;
     case 2: cout << “This month has 28 or 29 days”; break;
     case 4:
     case 6:
     case 9:
     case 11: cout << “This month has 30 days”; break;
     default: cout << “Invalid input”;
} // end switch
}
```

There are many cases of fall-through in this example.

A common error while using `switch` cases is to forget to put a **`break`** after the relevant block.

If `month == 1`, then `“This month has 31 days”` will be printed and the `switch` loop will exit due to the `break` statement. If there were no `break` after the `cout` statements here, then the other messages `“This month has 28 or 29 days”`, `“This month has 30 days”` and `“Invalid input”` will also be printed.

If `month == 11`, then `“This month has 30 days”` will be printed.

The `switch` case: a chess example

- The different pieces in a chess game are:

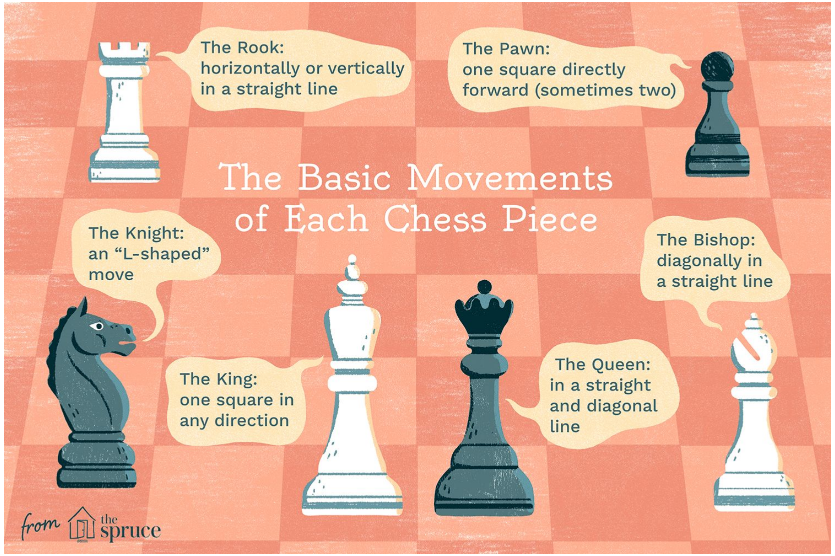

The Basic Movements of Each Chess Piece
The Rook: horizontally or vertically in a straight line
The Pawn: one square directly forward (sometimes two)
The Knight: an "L-shaped" move
The Bishop: diagonally in a straight line
The King: one square in any direction
The Queen: in a straight and diagonal line
from the spruce

Image source

Let us consider that each chess piece is identified by a number: pawn (0), knight (1), bishop (2), rook (3), queen (4), king (5)

Given a piece identifier and its starting coordinates (x1,y1) for a chess move and ending coordinates (x2,y2) of the chess move, supplied by a user, write a C++ program to determine whether the movement of the piece from (x1,y1) to (x2,y2) was legal as per chess rules. For simplicity, assume that the piece under consideration was the **only** one on the chess board. Use the **switch** case. The coordinates are in the order (row,column). One example is on the next slide.

The `switch` case: a chess example

```
flag = false; // flag as to whether the movement is legal
(true) or illegal (false)

switch(piece_num){

case 2: if (abs(x1-x2) == abs(y1-y2) && x1!=x2) flag =
true; break; // bishop

case 3: if ( (x1==x2 && y1!=y2) || (y1==y2 && x1!=x2))
flag = true; break; // rook

}
```

Convince yourself that the code snippet for bishop and rook is accurate and write code for other pieces.

`abs` is a function which computes the absolute value of `z` after the call `abs(z)`. To use it, you must include `stdlib.h` via `#include<stdlib.h>` in the beginning of the program after `#include<iostream>`

Ternary conditional operator

- C++ provides another ternary (three-way) type of conditional operator.
- It is compact but confusing. It is best to **avoid** it in programming (just my opinion), but you should know it.
- Form: `condition ? consequent : alternate ;`
- It is a compact equivalent to: `if (condition == true) consequent; else alternate;` or simply `if (condition) consequent; else alternate;`
- Example: `char grade = (score >= 35) ? ‘p’: ‘f’;` equivalent to `if (score >= 35) grade = ‘p’; else grade = ‘f’;`

Compound conditional expressions

- A compound expression with `&&` will be `true`, if and only if both (rather all) individual conditionals are `true`.
- A compound expression with `||` will be `true` if at least one of the individual conditionals is `true`.
- If `x` is `true`, then `x || false` is always `true`, and `!x` is `false`.
- DeMorgan's law #1: `!(x && y)` is the same as `!x || !y`
- DeMorgan's law #2: `!(x || y)` is the same as `!x && !y`

Watch out!

What will be the output of the following code snippets?

```
1)  if (0) cout << “Zero”;
2)  if (-1) cout << “Minus one”;
3)  if (1) cout << “One”;
4)  if (2) cout << “Two”;
5)  if (p = 0) { cout << “p is now zero”; } cout << p;
6)  if (true) cout << “True”;
7)  if (false) cout << “false”;
```