Chapter 2

Decision Making and Branching

Introduction

Decision making and branching let a program alter its execution path based on conditions. Without them, a program would always execute statements in fixed order (sequential). Real-world applications require choices: if user input satisfies some criteria, take one path; else take another. Branching statements make that possible. In many imperative languages (C, C++, Java), decision making statements include if, if-else, nested if-else, else if ladder, switch (or case), and historically goto (though discouraged).

Decision making has two aspects: evaluating conditions (Boolean expressions, relational and logical operators) and using branching statements to control flow.

Decision Making with IF Statement

The core is the if statement. You evaluate a condition; if it is true, you execute a block of statements; otherwise, skip them.

Syntax (in C style):

```
if (condition) {
   // statements
}
```

- condition is an expression that evaluates to true (nonzero) or false (zero).
- If true: execute statements inside {}.
- If false: skip to the statement following the if block.

Code example (C):

```
int x = 10;
if (x > 5) {
   printf("x is greater than 5\n");
}
printf("After if\n");
```

If x > 5 is true, "x is greater than 5" prints; otherwise, it is skipped.

Real-life example: In a banking app, you may check if account balance ≥ withdrawal amount. If yes, allow withdrawal; otherwise deny.

Simple IF Statement

The “simple if” is just a single condition and no alternative path.

As above, it is used when we only need to do something *if* a condition holds; if not, do nothing special.

Example (C):

```
if (temperature < 0) {
   printf("Freezing temperature\n");
}
```

If temperature is less than zero, you warn; else you just continue with the rest.

The IF-ELSE Statement

You often need to handle both the true and false cases. That’s the if-else structure.

Syntax:

```
if (condition) {
   // statements if true
```

```
} else {
   // statements if false
}
```
Example:
```
int age = 17;
if (age >= 18) {
   printf("Eligible to vote\n");
} else {
   printf("Not eligible to vote\n");
}
```
One and only one branch executes: either the if branch or the else branch.
Real-life example: If user's login credentials are valid, show dashboard; else show error message.

Nesting of IF-ELSE Statements
You can place an if or if-else inside another to allow hierarchical decisions.
Syntax:
```
if (condition1) {
   // block1
   if (condition2) {
      // block2
   } else {
      // block3
   }
} else {
   // block4
}
```
Example:
```
int x = 10;
if (x > 0) {
   printf("x positive\n");
   if (x % 2 == 0) {
      printf("x is even\n");
   } else {
      printf("x is odd\n");
   }
} else {
   printf("x is non-positive\n");
}
```
Here, only if x > 0 does the nested check apply. If x is non-positive, that nested logic is skipped.
Real-life example: In a form submission, first check if user is logged in; if yes, then check their permissions; else reject.

The ELSE IF Ladder
When there are multiple mutually exclusive conditions, an else-if ladder is used instead of deep nesting.
Syntax:
```
if (cond1) {
   // block1
```

```
} else if (cond2) {
   // block2
} else if (cond3) {
   // block3
} else {
   // default block
}
```

Example:

```
int marks = 78;
if (marks >= 90) {
   printf("Grade A\n");
} else if (marks >= 80) {
   printf("Grade B\n");
} else if (marks >= 70) {
   printf("Grade C\n");
} else {
   printf("Grade D or fail\n");
}
```

The conditions are tested top to bottom; first true branch executes, and rest are skipped (like a chain). If none match, else is run.

Real-life example: Tax slabs: if income > 1,000,000 then slab1, else if > 500,000 slab2, else slab3. This is better than nesting many ifs, easier to read.

The SWITCH Statement

switch (or case) provides a multi-way branching based on the value of an expression. It's often cleaner than many if-else-if when comparing a single variable to multiple constants.

Syntax in C:

```
switch (expression) {
   case const1:
      // statements
      break;
   case const2:
      // statements
      break;
   // ...
   default:
      // statements if no case matches
}
```

- expression is evaluated once.
- Control jumps to matching case label.
- Without break, execution "falls through" into subsequent cases (common in C).
- default handles cases not matched.
- You may group cases by omitting break (fall-through).

Example:

```
int day = 4;
switch (day) {
   case 1: printf("Monday\n"); break;
   case 2: printf("Tuesday\n"); break;
```

```
    case 3: printf("Wednesday\n");  break;
    case 4: printf("Thursday\n");  break;
    default: printf("Other  day\n");
}
```

If day == 4, prints “Thursday”.

Real-life example: Menu selection: user enters 1,2,3 to pick options. Use switch to branch to correct action.

Switch is more efficient and readable when many discrete cases exist.

The GOTO Statement

goto gives unconditional jump to another labeled line in the same function or block. It breaks structured control flow and is rarely recommended.

Syntax (C):

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

Example:

```
#include <stdio.h>

int main() {
    int n = 0;
    if (n == 0) {
        goto skip;
    }
    printf("This will be skipped if n == 0\n");
skip:
    printf("Jumped  or next\n");
    return 0;
}
```

If n == 0, the goto skip; directs execution to the skip: label, bypassing the print above.

Real-life example: In error handling deep in nested logic, you might jump to a cleanup section. But modern languages prefer structured error handling (exceptions, returns) instead.

Summary

- Decision making and branching let programs choose paths based on conditions.
- if handles a single conditional execution.
- if-else gives two-way branching (true vs false).
- Nested if-else supports hierarchical conditions.
- else if ladder is cleaner for many alternatives.
- switch enables multi-way branching on discrete values; supports fall-through in C.
- goto is an unconditional jump; valid but discouraged in structured programming.

These constructs are foundational to writing logic that adapts to inputs, handles multiple cases, and manages flow.

Decision Making and Looping

This section merges decision with iteration. Many real-world tasks require repeating work, with decisions inside loops. Constructs include while, do (or do-while), for, and the jump modifiers break, continue.

Introduction

Looping is repetition: execute a block multiple times until some condition is met. Combining loops with decision making yields powerful control—skip some iterations, exit early, or conditionally execute parts within loop. Most languages (C, Java, others) provide while, do-while (or do), for loops, plus control modifiers break and continue.

The WHILE Statement

while repeats a block as long as a condition is true. Condition is checked at the start of each iteration; if false initially, loop body may never execute.

Syntax (C-style):

```
while (condition) {
   // statements
}
```

Example:

```
int i = 0;
while (i < 5) {
   printf("i = %d\n", i);
   i++;
}
```

This prints from i = 0 to 4.

Real-life example: Read user input until they enter a valid value. Or in a GUI, poll until a user presses "exit".

The DO Statement (do-while)

do-while (or do loop) ensures the loop body executes at least once, since condition is tested after the body.

Syntax:

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

Example:

```
int i = 0;
do {
   printf("i = %d\n", i);
   i++;
} while (i < 5);
```

Outputs same as above.

Real-life example: In a menu-driven console app: display menu and ask user choice at least once, then repeat until they choose exit.

Many languages support do-while (C, C++, Java). Some languages (e.g. Python) don't support it natively; one simulates via while True and a break.

The FOR Statement

for loop is typically used when number of iterations is known or iterable structure available.

Syntax (C-style):

```
for (initialization; condition; update) {
   // statements
```

}

- initialization runs once at start.
- condition tested before each iteration.
- update runs at end of each iteration.

Example:

```
for (int i = 0; i < 5; i++) {
   printf("i = %d\n", i);
}
```

This also prints from 0 to 4.

Alternate usage (iterables in languages like Python):

```
for item in list_items:
   process(item)
```

Real-life example: Loop through array of orders and process each, or count from 1 to N for repeated tasks.

for is compact and clear when iteration count is known or data is collection.

Break and Continue Statements

Inside loops, sometimes you want to modify the flow: stop entirely or skip part of an iteration.

Break

break exits the nearest loop immediately. Execution continues with statement after loop.

Usage example:

```
for (int i = 0; i < 10; i++) {
   if (i == 5) {
      break;
   }
   printf("%d\n", i);
}
// prints 0 1 2 3 4
```

If you detect a condition where further looping is pointless (e.g. found desired element), use break.

Real-life example: Searching for a record in a list and stopping once found.

In nested loops, break only exits the inner loop; to exit outer loops, one may use flags or structured control or goto.

Continue

continue skips the rest of the current iteration and moves to the next iteration (re-evaluates loop condition).

Example:

```
for (int i = 0; i < 5; i++) {
   if (i == 2) {
      continue;
   }
   printf("%d\n", i);
}
// prints 0 1 3 4
```

Use continue to skip processing for certain cases inside loop without breaking entirely.

Real-life example: Iterating through transactions, skip invalid ones but continue processing valid ones.

Combined example

```
#include <stdio.h>

int main() {
   int data[] = {3, -1, 5, 0, 7};
   int n = 5;

   for (int i = 0; i < n; i++) {
      if (data[i] < 0) {
         // skip negative values
         continue;
      }
      if (data[i] == 0) {
         // stop entirely on zero
         break;
      }
      printf("Value: %d\n", data[i]);
   }

   return 0;
}
```

Suppose data[] = {3, -1, 5, 0, 7}:

- i = 0: data[0] = 3 → prints "Value: 3"
- i = 1: data[1] = -1 → continue skips and moves to next
- i = 2: data[2] = 5 → prints "Value: 5"
- i = 3: data[3] = 0 → break, loop ends
- Result: prints 3, 5, then stops.

Real-life: reading log entries, skip corrupt ones (continue), but if you hit a sentinel value (0) meaning "end", stop (break).