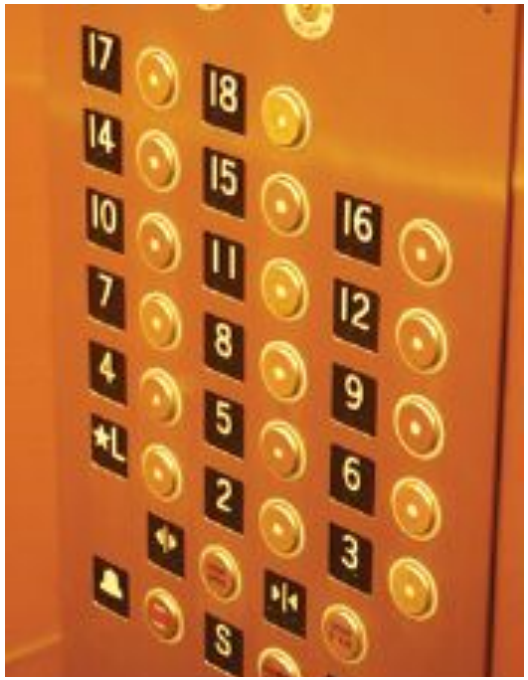
Topic 8

1. The `if` statement
2. Comparing numbers and strings
3. Multiple alternatives
4. Nested branches
5. Problem solving: flowcharts
6. Problem solving: test cases
7. Boolean variables and operators
8. Application: input validation
9. Chapter summary

Input Validation with `if` Statements

- Let's return to the elevator program to consider input validation.



- Assume that the elevator panel has buttons labeled 1 through 20 (*but not 13!*).
- Possible illegal inputs:
  - The number 13
  - Zero or a negative number
  - A number larger than 20
  - A value that is not a sequence of digits, such as five
- In each of these cases, we will give an error message and exit the program.

Input Validation with `if` Statements: Code

- It is simple to guard against an input of 13, or outside the range of floors:

```
if (floor == 13)
{
   cout << "Error: "
      << " There is no thirteenth floor."
      << endl;
   return 1;
}

if (floor <= 0 || floor > 20)
{
   cout << "Error: "
      << " The floor must be between 1 and 20."
      << endl;
   return 1;
}
```

Using `return` to exit the program upon Error

- The statement:

```
return 1;
```

  immediately exits the **`main`** function and therefore terminates the program.

- It is a convention to return the value 0 if the program completes normally, and a non-zero value when an error is encountered.

Input Validation: `cin.fail()`

- What if the user does not type a number in response to the prompt?

  'F' 'o' 'u' 'r' is not an integer response.

- When

  ```
  cin >> floor;
  ```

  is executed, and the user types in a bad input, the integer variable **`floor`** is not set.

- Instead, the input stream **`cin`** is set to a failed state.

Example of `cin.fail()`

- You can call the **`fail()`** member function to test for that failed state.

- So you can test for bad user input this way:

```
if (cin.fail())
{
   cout << "Error: Not an integer." << endl;
   return 1;
}
```

In a later chapter, we will explain how to clear the failed state, so further input can be taken.

Input Validation with `if` Statements – Elevator Program

ch03/elevator2.cpp

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

int main()
{
   int floor;
   cout << "Floor: ";
   cin >> floor;

   // The following statements check various input errors
   if (cin.fail())
   {
      cout << "Error: Not an integer." << endl;
      return 1;
   }
   if (floor == 13)
   {
      cout << "Error: There is no thirteenth floor." << endl;
      return 1;
   }
   if (floor <= 0 || floor > 20)
   {
      cout << "Error: The floor must be between 1 and 20." << endl;
      return 1;
   }
```

Input Validation with `if` Statements – Elevator Program, Pt.2

```
   // Now we know that the input is valid
   int actual_floor;
   if (floor > 13)
   {
      actual_floor = floor - 1;
   }
   else
   {
      actual_floor = floor;
   }

   cout << "The elevator will travel to the actual
  floor "
      << actual_floor << endl;

   return 0;
}
```

Topic 9

1. The `if` statement
2. Comparing numbers and strings
3. Multiple alternatives
4. Nested branches
5. Problem solving: flowcharts
6. Problem solving: test cases
7. Boolean variables and operators
8. Application: input validation
9. Chapter summary

Chapter Summary Part #1

The `if()` statement implements a decision.

- The `if()` statement allows a program to carry out different actions depending on the nature of the data to be processed.

Implement comparisons of numbers and objects.

- Relational operators (`< <= > >= == !=`) are used to compare numbers and strings.
- Lexicographic order is used to compare strings.
  - "car" is less than "cart"

Complex decisions require multiple `if()`...`else` statements.

- Multiple alternatives are required for decisions that have more than two cases.
- When using multiple `if()` statements, pay attention to the order of the conditions.

Chapter Summary Part #2

Implement decisions whose branches require further decisions.

- When a decision statement is contained inside the branch of another decision statement, the statements are *nested*.
- Nested decisions are required for problems that have >=2 levels of decision making, such as the tax code.

Draw flowcharts to visualize control flow in a program.

- Flow charts are made up of elements for tasks, input/outputs, and decisions.
- Each branch of a decision can contain tasks and further decisions.
- Never point an arrow inside another branch.

Chapter Summary Part #3

Design test cases for your programs.

- Each branch of your program should be tested.
- Design test cases **before** implementing a program.

Use the `bool` data type to store and combine conditions that can be `true` or `false`.

- C++ has two Boolean operators that combine conditions: `&&` (*and*) and `||` (*or*).
- To invert a condition, use the **!** (*not*) operator.
- The `&&` and `||` operators use *short-circuit evaluation*: As soon as the value is determined, no further conditions are evaluated.
- Use De Morgan's law to simplify combinations:
  `!(A && B)` is the same as `!A || !B`
  `!(A || B)` is the same as `!A && !B`

Chapter Summary Part #4

Apply `if()` statements to detect whether input is valid.

- When reading a value, check that it is within the required range.
- Use the `fail()` function to test whether the input stream has failed:

```
if (cin.fail())
{
   cout << "Error: Not an integer." << endl;
   return 1;
}
```