Data type bool
Integer type bool represents the boolean values true and false. The bool value false has an integer value of 0, and bool value true has an integer value of 1.

Example 1:
bool a = true, b = false, c = false;  // declarations and initialization of 3 boolean variables

Example 2:
bool flag;  // declare the boolean variable “flag” (flag can have value “true” or “false”)  
flag = (3 < 5);  // flag = true, since 3 is indeed less than 5 (i.e., the expression 3 < 5 is true).

Any expression whose value is either true or false is called a boolean expression or a logical expression. 
Examples: (3 < 5), (x > y), (z > x*x + y) are all boolean expressions.

Boolean expressions and variables are most frequently used with relational and logical operators.

Relational operators
Can be used to compare integer and floating-point expressions.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Example</th>
<th>true if</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equal</td>
<td>x == y</td>
<td>x and y are equal</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
<td>x != y</td>
<td>x and y are not equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>x &gt; y</td>
<td>x is greater than y</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>x &gt;= y</td>
<td>x is greater than or equal to y</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>x &lt; y</td>
<td>x is less than y</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>x &lt;= y</td>
<td>x is less than or equal to y</td>
</tr>
</tbody>
</table>

Logical operators
x and y could be boolean expressions or variables here.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Example</th>
<th>true if</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>AND</td>
<td>x &amp;&amp; y</td>
<td>x and y are both true</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OR</td>
</tr>
<tr>
<td>!</td>
<td>NOT</td>
<td>!x</td>
<td>x is false</td>
</tr>
</tbody>
</table>

Example 1:
bool a = true, b = false, c = false; 
Then expression (a && !b) || c is equivalent to (true && true) || (false) = true || false = true

Example 2:  (2 > 3) && (27 < 99) = (false) && (true) = false
Control Flow

\texttt{if} statement

the \texttt{if} statement is one of the most powerful devices in programming. It allows the program to choose an action based on input. To execute an \texttt{if} statement, we first determine if some logical (or boolean) expression is true or false. If the expression is true, we execute the action and proceed to the statement immediately following the action. But, if the expression is false, we do not execute the action; we skip it and proceed to the statement immediately following. Note that the logical expression must be enclosed in \textit{parentheses}.

\begin{verbatim}
if (logical expression)  
  action;  // here, \textit{action} consists of one statement without braces.  
  // this is called the \textbf{true branch}  
  // if we want multiple actions to occur when expression is true,  
  // we must use braces!

next statement;

if (logical expression) {  
  action 1;  // here, the true branch consists of \textit{multiple} statements  
  action 2;  // therefore, we must use braces
}
next statement;  // if the expression is false, this is the first statement that is executed
\end{verbatim}

Example:

\begin{verbatim}
int code = 2;  
if (code != 2) {  
  cout << “The water was too warm” << endl;  
  cout << “The waters were all fished out” << endl;
}

cout << “*** End of fishing excuses ***” << endl;
\end{verbatim}

Then the output is:

*** End of fishing excuses ***

\texttt{if-else} statement

\begin{verbatim}
if (logical expression)  
  action 1a;  // this is the \textbf{true branch}. If expression is false, we skip this.  
  // again, if we want multiple actions to occur when expression is true,  
  // we must use braces!

else  
  action 1b;  // this is the \textbf{false branch}. if expression is true, we skip this.

next statement;  // this is the 2\textsuperscript{nd} statement executed (whether expression is true or false)
\end{verbatim}
nested if-else statements

if (expression_1)
    action 1; // again, use braces anywhere you have multiple actions
else if (expression_2)
    action 2;
else if (expression_3)
    action 3;
...
else if (expression_n)
    action n;
else
    default action;
next statement;

Here, the expressions are evaluated in order (expression_1, expression_2, etc.) until one (if any!) is found to be true. If expression_i is the first true expression, then action_i is executed followed by next statement; all other actions are skipped, and no other expressions are evaluated.

If no expressions are true, then default action is executed followed by next statement.

COMMON ERROR: putting a semicolon after the if statement. Results of this error:
• The true branch becomes a null statement
• The statement following the semicolon would be executed regardless of true/false expression
• The else would cause a compile error – it is no longer associated with any if statement.

Example with nested if-else statements:

#include <iostream>
using namespace std;

int main()
{
    int age; // must declare a variable
    cout << “Please enter your age:” << endl; // asks for age
    cin >> age; // input is stored in variable “age”
    if (age < 100)
        cout << “You are pretty young!”;
    else if (age == 100)
        cout << “You are old”; // case age = 100
    else if (age >100)
        cout << “You are really old”; // case age is greater than 100
    return 0;
}
**while statement**

while loops are very simple. The basic structure is:

```
while (expression)
    action; // again, for multiple actions, use braces
next statement;
```

So while expression is true, then execute all the code in the loop. In other words, if expression is true, we execute the action and return to the top of the loop. We again test the expression; if expression is true, we execute the action and return to the top. Keep repeating this process until the expression is false; when expression is false, we skip to the next statement.

Example: What is the output?

```
int x;
x = 7;
while (x >= 0) {
    x = x - 2;
    cout << x << endl; // print current value of x, goto next line
}
```

Output:

```
5
3
1
-1
```

**do while statement**

do while loops are useful for things that we want to loop at least once. The structure is:

```
do
    action;
while (expression);
next statement;
```

Same as while statement, but now we test the expression at the bottom of the loop.

Example:

**do while** can be used to verify user input:

```
int response;
do {
    cout << “Enter a positive integer: “; // asks user to input a positive integer
    cin >> response;
} while (response < 0); // if input is invalid (negative), the user is asked again
```
**for statement**

For loops are the most useful type of loop. They are particularly useful for counted loops (loops that execute for a predetermined number of times). The structure is:

```
for (expr1; expr2; expr3)
  action;
next statement;
```

- **expr1** is used to initialize the loop variables.
- **expr2** is used to test whether to execute **action**. (**expr2** is a logical or boolean expression)
- **expr3** is used to update the loop counter.
- **action** is the body of the loop (i.e., the statement(s) executed in each loop iteration).

This structure of the **for** loop is logically equivalent to:

```
expr1;
while (expr2) {
  action;
  expr3;
}
```

Example:

```
for (i = 0; i < 10; i = i + 1)
  cout << i << " " ;  // prints current value of i followed by a space. stays on same line.
```

Output =
0 1 2 3 4 5 6 7 8 9

Example: What is the error here?

```
for (i = 1, i <=5, i = i + 1)
  cout << i << endl;
```

Answer: SEMICOLONS, not commas, must separate the three expressions in the **for** statement.