Recall Standard C++ supports a simple data type specialized for representing logical values. 

boolean type variables can have either of two values: true or false

The identifiers true and false are C++ reserved words.

In C++, in order to ask a question, a program makes an assertion which is evaluated to either true or false at run-time.

In order to assert "The student's age is above or equal to 21?", in C++:

```cpp
const int LEGALAGE = 21;
bool isLegalAge;
int stuAge;
cin >> stuAge;
isLegalAge = (stuAge >= LEGALAGE );
```

The value of isLegalAge can now be tested to see if it is true or false.
Boolean expressions can, generally, take one of two forms.

The first is a relational expression, an expression (e.g., arithmetic) followed by a relational operator followed by another expression.

For example: \( (b \times b - 4 \times a \times c) > 0 \)

C++ has six standard relational operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equals</td>
</tr>
<tr>
<td>!=</td>
<td>does not equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>is greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>is greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>is less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>is less than or equal to</td>
</tr>
</tbody>
</table>

The relational operators can be used to compare two values of any of the built-in types discussed so far.

Most mixed comparisons are also allowed, but frequently make no sense.
Relational Expression Examples

Given:

```cpp
const int MAXSCORE = 100;
char MI = 'L', MI2 = 'g';
int Quiz1 = 18, Quiz2 = 6;
int Score1 = 76, Score2 = 87;
string Name1 = "Fred", Name2 = "Frodo";
```

Evaluate:

```cpp
Quiz1 == Quiz2
Score1 >= Score2
Score1 > MAXSCORE
Score1 + Quiz1 <= Score2 + Quiz2

MI == MI2
MI < MI2
'Z' < 'a'

Name1 < Name2
```
A logical expression consists of a Boolean expression followed by a Boolean operator followed by another Boolean expression (with negation being an exception).

C++ has three Boolean (or logical) operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>not</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Boolean operators && and || are binary, that is each takes two operands, whereas the Boolean operator ! is unary, taking one operand.

The semantics of the Boolean operators are defined by the following "truth tables":

<table>
<thead>
<tr>
<th>A</th>
<th>!A</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A &amp;&amp; B</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

| A   | B   | A || B |
|-----|-----|-------|
| true| true| true  |
| true| false| true  |
| false| true| true  |
| false| false| false |
Logical Expression Examples

Given:

```cpp
const int MINHEIGHT = 42, MAXHEIGHT = 54;
int FredsHeight, AnnsHeight;
int EmmasHeight = 45;
```

Evaluate:

```cpp
MINHEIGHT <= EmmasHeight && EmmasHeight <= MAXHEIGHT
! (EmmasHeight > MAXHEIGHT)

// When would the following be true? false?
FredsHeight < MINHEIGHT || FredsHeight > MAXHEIGHT
```

Two Boolean expressions are logically equivalent if they are both true under exactly the same conditions. Are the following two Boolean expressions logically equivalent?

```cpp
!(EmmasHeight > FredsHeight)
EmmasHeight < FredsHeight
```
DeMorgan's Laws

Suppose that A and B are logical expressions. Then DeMorgan's Laws state that:

\[ \neg (A \land B) \iff (\neg A) \lor (\neg B) \]
\[ \neg (A \lor B) \iff (\neg A) \land (\neg B) \]

The Principle of Double Negation states that:

\[ \neg (\neg A) \iff A \]

(The symbol \(\iff\) indicates logical equivalence.)

So the following negation:

\[ \neg(Fred\text{'}s\text{ }\text{Height} < \text{MINHEIGHT} \lor Fred\text{'}s\text{ }\text{Height} > \text{MAXHEIGHT}) \]

...could be rewritten equivalently as:

\[ (Fred\text{'}s\text{ }\text{Height} \geq \text{MINHEIGHT} \land Fred\text{'}s\text{ }\text{Height} \leq \text{MAXHEIGHT}) \]
Since Boolean expressions can involve both arithmetic and Boolean operators, C++ defines a complete operator evaluation hierarchy:

0. Expressions grouped in parentheses are evaluated first.
1. (unary) – !
2. * / %
3. + –
4. <= >= < >
5. == !=
6. &&
7. ||
8. =

Operators in groups (2) thru (7) are evaluated left to right, but operators in groups (1) and (8) are evaluated right to left.
Boolean Examples

Given:

\[
\text{int } i = 3, k = 5, \\
j = 0, m = -2;
\]

Evaluate:

\[
(0 < i) \land (i < 5) \\
(i > k) \lor (j < i) \\
! (k > 0)
\]

\[
3 \times i - 4 / k < 2 \\
i + j < k \\
(i > 0) \land (j < 7) \\
(i < k) \lor (j < 7) \\
(m > 5) \lor (j > 0)
\]

Gotcha's:

\[
k = 4 \quad \text{// confusing equality and assignment}
\]

\[
0 < i < 2 \quad \text{// allowed, but. . . it doesn't} \\
\quad \text{// mean what you think. . .}
\]
Flow of Execution: the order in which the computer executes statements in a program.

Default flow is sequential execution:

```cpp
cin >> x;
y = x * x + x + 1;
cout << y << endl;
```

Control structure: a statement that is used to alter the default sequential flow of control

Selection: a control structure that allows a choice among two or more actions
Selection: \textit{if} Statement

The simplest selection structure in C++ is the \textit{if} statement. Syntactically:

\begin{center}
\begin{tikzpicture}
  \node (if) [fill, circle, draw] {\textit{if}};
  \node (expr) [right=of if] {<boolean expression>};
  \node (clause) [right=of expr] {<if-clause>};
  \draw [->] (if) -- (expr);
  \draw [->] (expr) -- (clause);
\end{tikzpicture}
\end{center}

The Boolean expression must be enclosed in parentheses, and \textit{<if-clause>} can be a single C++ statement or a compound statement.

The semantics of the \textit{if} statement are:

\begin{center}
\begin{tikzpicture}
  \node (if) [fill, circle, draw] {\textit{if}};
  \node (expr) [right=of if] {<boolean expression>};
  \node (true) [below=of expr] {true};
  \node (false) [right=of true] {false};
  \node (clause) [right=of true] {<if-clause>};
  \draw [->] (if) -- (expr);
  \draw [->] (expr) -- (true);
  \draw [->] (true) -- (clause);
  \draw [->] (expr) -- (false);
\end{tikzpicture}
\end{center}

The \textit{if} statement is used to select between performing an action and not performing it:

```cpp
if (Grade == 'A') {
    cout << "Good Job!";
}
```
C++ also provides a selection structure for choosing between two alternatives, the `if...else` statement. Syntactically:

```c++
if (Grade == 'A') {
    cout << "Good job!";
} else {
    cout << "Grades aren't everything."
    << endl;
    cout << "But they help."
}
```

Semantically:

The `if...else` construct allows making an either-or choice:

```c++
if (Grade == 'A') {
    cout << "Good job!";
} else {
    cout << "Grades aren't everything."
    << endl;
    cout << "But they help."
}
The if-clause and else-clause may contain any valid C++ statements, including other if or if...else statements:

```cpp
const double LOFLOOR = 100.0;
const double HIFLOOR = 500.0;
const double LORATE = 0.05;
const double HIRATE = 0.10;
double orderAmt;

... 
if (orderAmt <= LOFLOOR) {
    Discount = 0.0;
}
else {
    if (orderAmt <= HIFLOOR) {
        Discount = LORATE * (orderAmt - LOFLOOR);
    }
    else {
        Discount = 20.0 +
                HIRATE * (orderAmt - HIFLOOR);
    }
}
```

Conditions that are "mutually exclusive", (one condition being true excludes all others from being true), should be tested for with nested ifs, (as opposed to disjoint ifs), for efficiency.
Deeper Nesting

In some cases a problem may require a relatively large number of nested layers. In that case, the formatting used on the previous slide would cause the code to be poorly formatted. An alternative:

```cpp
cout << "Your semester grade is ";

if (Average >= 90)  // Note the layout and indenting style.
    cout << "A" << endl;
else if (Average >= 80)
    cout << "B" << endl;
else if (Average >= 70)
    cout << "C" << endl;
else if (Average >= 60)
    cout << "D" << endl;
else
    cout << "F" << endl;
```
Simple Sorting

Given three int variables (a, b, c), having distinct values, output the values in descending order:

```cpp
if (a > b) {  // Get order of a and b;
    // if clause if a is larger
    if (a > c) {  // a is largest; now
        // sort out b and c
        if (b > c)
            cout << a << b << c; // c is smallest
        else
            cout << a << c << b; // c is middle
    }
    else
        cout << c << a << b; // c is largest
}  // else clause if b is larger
else {
    // b is largest; now
    if (b > c) {  // b is largest; now
        // sort out a and c
        if (a > c)
            cout << b << a << c; // c is smallest
        else
            cout << b << c << a; // c is middle
    }
    else
        cout << c << b << a; // c is largest
}
```
Using nested if and if...else statements raises a question: how can you determine which if an else goes with?

The syntax rule is simple: an else is paired with the closest previous uncompleted if.

The correct interpretation of the code above would be clearer if the programmer had used braces to group statements (even though none are necessary). Consider:

What do you think the programmer intended here?

Does this achieve it?

How could it be improved?
```cpp
#include <iostream>
using namespace std;

int main() {
    const int GREGORIAN = 1752;
    int Year;
    bool yearDivisibleBy4, yearDivisibleBy100, yearDivisibleBy400;

    cout << "This program determines if a year of the "
    << "Gregorian calendar is a leap year."
    << endl;
    cout << "Enter the possible leap year: ";
    cin >> Year; // 1

    if (Year < GREGORIAN) {
        cout << endl << "The year tested must be on the "
        << "Gregorian calendar." << endl;
        cout << "Reenter the possible leap year: ";
        cin >> Year; // 2
    } // end of if (Year < GREGORIAN)

    . . .
```
Example Program

```cpp
... 
yearDivisibleBy4 = (( Year % 4 ) == 0); // 3
yearDivisibleBy100 = (( Year % 100 ) == 0); // 4
yearDivisibleBy400 = (( Year % 400 ) == 0); // 5

if ( ((yearDivisibleBy4) && (! yearDivisibleBy100)) || // 6
    (yearDivisibleBy400) )
    cout << "The year " << Year << " is a leap year." << endl;
else
    cout << "The year " << Year << " is NOT a leap year." << endl;

return 0;
}
```
### Execution Trace

Execution Trace (Desk-Checking) - hand calculating the output of a program with test data by mimicking the actions of the computer.

<table>
<thead>
<tr>
<th>Year</th>
<th>yearDivisibleBy4</th>
<th>yearDivisibleBy100</th>
<th>yearDivisibleBy400</th>
<th>Boolean expr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Although tedious, execution tracing can detect many logic errors early in the process.

Note that this same organized procedure can be applied to an algorithm as easily as to code.
Massively Multiple Selections

Some problems require making simple choices among a large number of alternatives.

For instance, consider this simple code fragment for encrypting numbers:

```cpp
In.get(nextCharacter);
while (In) {
    if (nextCharacter == '0')
        cout << '3';
    else if (nextCharacter == '1')
        cout << '7';
    else if (nextCharacter == '2')
        cout << '5';
    else if (nextCharacter == '3')
        cout << '0';
    else if (nextCharacter == '4')
        cout << '9';
    else if (nextCharacter == '5')
        cout << '2';
    else if (nextCharacter == '6')
        cout << '8';
    else if (nextCharacter == '7')
        cout << '6';
    else if (nextCharacter == '8')
        cout << '1';
    else if (nextCharacter == '9')
        cout << '4';
    else
        cout << nextCharacter;
    In.get(nextCharacter);
}
```

The code is not difficult, but it is repetitive and ugly.

C++ provides an alternative selection structure that is an improvement in this situation.
The C++ switch statement may be used to replace a nested if...else when the comparisons are all for equality, and the compared values are characters or integers:

```
switch ( <selector> ) {
    case <label 1>: <statements 1>; break;
    case <label 2>: <statements 2>; break;
    ..
    case <label n>: <statements n>; break;
    default: <statements d>
}
```

- `<selector>` - a variable or expression of type `char` or `int`
- `<label i>` - a constant value of type `char` or `int`
- labels cannot be duplicated

When the switch statement is executed, the selector is evaluated and the statement corresponding to the matching constant in the unique label list is executed. If no match occurs, the default clause is selected, if present.

The type of selector must match the type of the constants in the label lists.
Encryption Example Revisited

Here is the encryption algorithm implemented with a `switch` statement:

```cpp
while (In) {
    switch (nextCharacter) {
        case '0': cout << '3'; break;
        case '1': cout << '7'; break;
        case '2': cout << '5'; break;
        case '3': cout << '0'; break;
        case '4': cout << '9'; break;
        case '5': cout << '2'; break;
        case '6': cout << '8'; break;
        case '7': cout << '6'; break;
        case '8': cout << '1'; break;
        case '9': cout << '4'; break;
        default: cout << nextCharacter;
    }
    In.get(nextCharacter);
}
```

The logical effect is the same, but...

- this code is easier to read.
- this code will execute slightly faster.
- this code may be easier to modify.
Switch Details

If the selector value does not match any case label, and there is no default case, then execution simply proceeds to the first statement following the end of the switch.

If a case clause omits the break statement, then execution will "fall through" from the end of that case to the beginning of the next case.

It is legal for a case clause to be empty.

```cpp
switch (LetterGrade) {
    case 'A': cout << "very ";
    case 'B': cout << "good job";
        break;
    case 'C': cout << "average";
        break;
    case 'I':
    case 'D': cout << "danger";
        break;
    case 'F': cout << "failing";
        countF = countF + 1;
        break;
    default: cout << "Error: invalid grade";
}
```
A switch statement can only be used in cases involving an equality comparison for a variable that is of integral type (i.e., char or int).

Therefore, a switch cannot be used when checking values of a float, double or string variable.

```c++
... 
if (Command == "add") {
    Result = leftOp + rightOp;
}
else if (Command == "mult") {
    Result = leftOp * rightOp;
}
else if (Command == "sub") {
    Result = leftOp - rightOp;
}
else if (Command == "div" && rightOp != 0) {
    Result = leftOp / rightOp;
}
... 
```

Also, the nested if...else on slide 5.13 cannot be replaced with an equivalent switch statement because the decisions are based on inequality comparisons.
Short Circuiting

C++ is very economical when evaluating Boolean expressions. If in the evaluation of a
compound Boolean expression, the computer can determine the value of the entire
expression without any further evaluation, it does so. This is called short circuiting. What
does this mean for us?

```cpp
int main() {
    const int SENTINEL = 0;
    ifstream In("Heights.txt");

    int nextHeight;
    int totalHeight = 0;
    int numHeights = 0;

    while ( (In >> nextHeight) && (nextHeight > SENTINEL) ) {
        totalHeight = totalHeight + nextHeight;
        numHeights++;
    }

    if ( numHeights > 0 ) {
        cout << fixed << showpoint << setprecision(2);
        cout << double(totalHeight) / numHeights << endl;
    }
    In.close();
    return 0
}
```
In Standard C++, `bool` is a simple data type built into the language.

C++ variables declared as type `bool` can be used in the natural and obvious way.

In C, there is no Boolean type variable. Instead, integer values are used to represent the concepts of true and false. The convention is that 0 (zero) represents false, and that any nonzero value (typically 1) is interpreted as representing true.

Thus, in C, one might write the following (compare to slide 5.1):

```c
const int LEGALAGE = 21;
int isLegalAge; // Can have any int value.
isLegalAge = (stuAge >= LEGALAGE);
```

Now, the variable `isLegalAge` will have an integer value, interpreted as described.

C++ inherits the C-style treatment, so we could then still write:

```c
if (isLegalAge) cout << "OK";
else cout << "Nope";
```

The use of integer values as Booleans is poor programming style in C++. 