CONDITIONAL STATEMENTS

OVERVIEW

OVERVIEW

- **Many times we want programs to make decisions**
  - What drink should we dispense from the vending machine?
  - Should we let the user withdraw money from this account?

- **We make this choice by looking at values of variables**
  - When variables meet one condition we do one thing
  - When variables do not meet condition we do something else

- **To make decisions in a program we need <u>conditional statements</u> that let us take different paths through code**

OVERVIEW

- **In C++ there are three types of conditional statements:**
  - The if statement
  - The if-else statement
  - The switch statement

- **Lesson objectives:**
  - Learn how logical expressions are written
  - Learn the syntax and semantics of conditional statements
  - Study example programs showing their use
  - Complete online lab on conditional statements
  - Complete programming project using conditional statements

CONDITIONAL STATEMENTS

PART 1

LOGICAL EXPRESSIONS

LOGICAL EXPRESSIONS

- **The fundamental building block of all C++ conditional statements is the <u>logical expression</u>**
  - Logical expressions always return a Boolean value of either true or false
  - Logical expressions are used to decide what portions of the program to execute and what to skip over

- **Simple logical expressions are of the form:**

  (data relational_operator data)

  - Data terms in logical expressions can be variables, constants or arithmetic expressions

LOGICAL EXPRESSIONS

- **The C++ relational operators are:**

<	**less than**
>	**greater than**
<=	**less than or equal**
>=	**greater than or equal**
==	**equal to**
!=	**not equal to**

LOGICAL EXPRESSIONS

- **Examples using numbers:**
  - (17 < 42) is true
  - (42 > 17) is true
  - (17 == 42) is false
  - (42 != 17) is true
  - ((42 - 17) > (42 + 17)) is false
  - ((17 * 3) <= (17 + 17 + 17) is true

  - string str = “john”
  - (“JOHN” == str) is false
  - (“abc” < “xyz”) is true

LOGICAL EXPRESSIONS

- **Examples with variables:**
  - int a=17, b=42;
  - (a < b) is true
  - (a >= b) is false
  - (a == 17) is true
  - (a != b) is true
  - ((a + 17) == b) is false
  - ((42 – a) < b) is true

LOGICAL EXPRESSIONS

- **Warning: Do not use a single = for checking equality**
  - If you use = instead of == you will NOT get an error message but it will return a true/false value you are NOT expecting
  - The = operator is only used for data assignment to variables as we saw in the previous section

- **Warning: Do not use =<, =>, =! to compare data values**
  - You will get a compiler error message if you type these relational operators in backwards
  - Just remember the correct operators <=, >=, != all end with “equal” just like the phrases “less than or equal”

COMPLEX LOGICAL EXPRESSIONS

- **We can combine simple logical expressions to get complex logical expressions that are more powerful**
  - For example: checking the user has entered enough money AND the vending machine has that item available
- **The syntax is: (expression logical_operator expression)**
  - The two expressions above can either be simple logical expressions or complex logical expressions
- **The C++ logical operators are:**

&&	**and**
||	**or**

COMPLEX LOGICAL EXPRESSIONS

- **Truth tables are often be used to enumerate all possible values of a complex logical expression**
  - We make columns for all logical expressions
  - Each row illustrates one set of input values
  - The maximum number of rows is always a power of 2

A	B	A && B	A || B
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE

↑ (A && B): Only true if both A and B are true

↑ (A || B): Only true if either A and B are true

COMPLEX LOGICAL EXPRESSIONS

- **C++ evaluates complex logical expressions from left to right**
  - (exp1 && exp2) will be true if <u>both</u> exp are true
  - (exp1 && exp2 && exp3) will be true if <u>all</u> exp are true
  - (exp1 || exp2 || exp3) will be true if <u>any</u> exp is true

- **C++ has a feature called “conditional evaluation” that will stop the evaluation early in some cases**
  - (exp1 && exp2) will be false if exp1 is false
  - (exp1 || exp2) will be true if exp1 is true
  - In both cases, C++ does not need to evaluate exp2 because the answer is already known after looking at exp1

EXAMPLES

```
Num1 = 23;
Num2 = 0;
(Num2 > 0) && (Num1/Num2 < 42) ---> false
```

```
Num1 = 49;
Num2 = 7;
(Num2 > 0) && (Num1/Num2 < 42) ---> true
```

COMPLEX LOGICAL EXPRESSIONS

- **Complex logical expressions**
  - ((17 < 42) && (42 < 17)) is false, because second half is false
  - ((17 <= 42) || (42 <= 17)) is true, because first half is true

- **When float variables x = 3.14 and y = 7.89**
  - ((x < 4) && (y < 8)) is true, because both halves are true
  - ((x > 3) && (y > 8)) is false, because second half is false
  - ((x < 4) || (y > 8)) is true, because first half is true
  - ((x < 3) || (y < 8)) is true, because second half is true
  - ((x > 4) || (y > 8)) is false, because both halves are false

START HERE

THE NOT OPERATOR

- **The not operator in in C++ reverses the value of any logical expression**
  - Logically “not true” is same as “false”
  - Logically “not false” is same as “true”

- **The C++ syntax for the not operator is: ! expression**
  - This is a “unary” operator since there is just one logical expression to the right of the not operator

THE NOT OPERATOR

- **Examples with integer variables a = 7 and b = 3**

(a > b) is true	! (a > b) is false
(a <= b) is false	! (a <= b) is true
(a == b) is false	! (a == b) is true
(a != b) is true	! (a != b) is false

THE NOT OPERATOR

- **We can often "move the not operation inside" a simple logical expression**
- **To do this simplification, we need to remove the ! operator and "reverse the logic" of the relational operator**
  - ! (a < b) same as (a >= b)
  - ! (a <= b) same as (a > b)
  - ! (a > b) same as (a <= b)
  - ! (a >= b) same as (a < b)
  - ! (a == b) same as (a != b)
  - ! (a != b) same as (a == b)

Notice that
the opposite of < is >=
the opposite of > is <=
the opposite of == is !=

THE NOT OPERATOR

- **When exp1 and exp2 are simple logical expressions**
  - ! (exp1 && exp2) is same as (!exp1 || !exp2)
  - ! (exp1 || exp2) is same as (!exp1 && !exp2)
  - ! (!exp1 || !exp2) is same as (!!exp1 && !!exp2) or (exp1 && exp2)
  - ! (!exp1 && !exp2) is same as (!!exp1 || !!exp2) or (exp1 || exp2)
- **Hence, there are many different ways to represent the same logical expression**
  - Your goal when programming is to choose the simplest logical expression that represents the relationships you are looking for

THE NOT OPERATOR

- **Examples with float variables x = 4.3 and y = 9.2**
  - !((x < 5) && (y < 10)) is false
  - (!(x < 5) || !(y < 10)) is false
  - ((x >= 5) || (y >= 10)) is false

  - !((x >= 5) || (y >= 10)) is true
  - (!(x >= 5) && !(y >= 10)) is true
  - ((x < 5) && (y < 10)) is true

To most people, these logical expressions are the simplest to read and understand

SUMMARY

- **In this section, we have focused on how logical expressions can be written in C++**
- **We have seen how relational operators (<, <=, >, >=, ==, and !=) can be used to create simple logical expressions**
- **We have seen how logical operators (&& and !!) can be used to make more complex logical expressions**
- **Finally, we have seen how the not operator (!) can be used to reverse the true/false value of logical expressions**

DE MORGAN'S LAWS (OPTIONAL)

- **We can extend truth tables to study the not operator**
  - Add new columns showing !A and !B and their use in complex logical expressions with && and ||

A	B	!A	!B	A && B	A || B	!A && !B	!A || !B
TRUE	TRUE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE
FALSE	TRUE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE

Notice anything interesting here?

DE MORGAN'S LAWS (OPTIONAL)

- **We can extend truth tables to study the not operator**
  - Add new columns showing !A and !B and their use in complex logical expressions with && and ||

A	B	!A	!B	A && B	A || B	!A && !B	!A || !B
TRUE	TRUE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE
FALSE	TRUE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE

These columns have <u>opposite</u> values so
! (A || B) is the same as !A && !B

DE MORGAN'S LAWS (OPTIONAL)

- **We can extend truth tables to study the not operator**
  - Add new columns showing !A and !B and their use in complex logical expressions with && and ||

A	B	!A	!B	A && B	A || B	!A && !B	!A || !B
TRUE	TRUE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE
FALSE	TRUE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE

A similar pattern occurs here too

DE MORGAN'S LAWS (OPTIONAL)

- **We can extend truth tables to study the not operator**
  - Add new columns showing !A and !B and their use in complex logical expressions with && and ||

A	B	!A	!B	A && B	A || B	!A && !B	!A || !B
TRUE	TRUE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE
FALSE	TRUE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE

These columns have <u>opposite</u> values so
! (A && B) is the same as !A || !B

DE MORGAN'S LAWS (OPTIONAL)

- **From the truth tables above we saw:**

  ! (A || B) is the same as !A && !B

  "not (A or B)" is the same as "(not A) and (not B)"

  ! (A && B) is the same as !A || !B

  "not (A and B)" is the same as "(not A) or (not B)"

- **These rules are known as "De Morgan's Laws"**
  - We can use this rule to simplify a complex logical expression by "moving the not operation inside"
  - We can also simplify !A and !B by "reversing the logic" of the relational operator
  - The final result is a statement that is logically equivalent to the initial statement and often easier to read / understand

DE MORGAN'S LAWS (OPTIONAL)

- **To apply De Morgan's Laws, we must change the logical operator and the expressions**
  - The && operator changes into ||
  - The || operator changes into &&
  - The ! is applied to both expressions

- **Two not operators side by side cancel each other out so they can be removed without changing the expression**
  - "!! true" is equal to "! false" which is equal to "true"

CONDITIONAL STATEMENTS

PART 2

IF STATEMENTS

THE IF STATEMENT

- **Sometimes we want to selectively execute a block of code**
- **The C++ syntax of the if statement is:**

```
if ( logical expression )
{
   // Block of code to execute if expression is true
}
```

- **When expression is <u>true</u>, the block of code is executed**
- **When expression is <u>false</u>, the block of code is skipped**

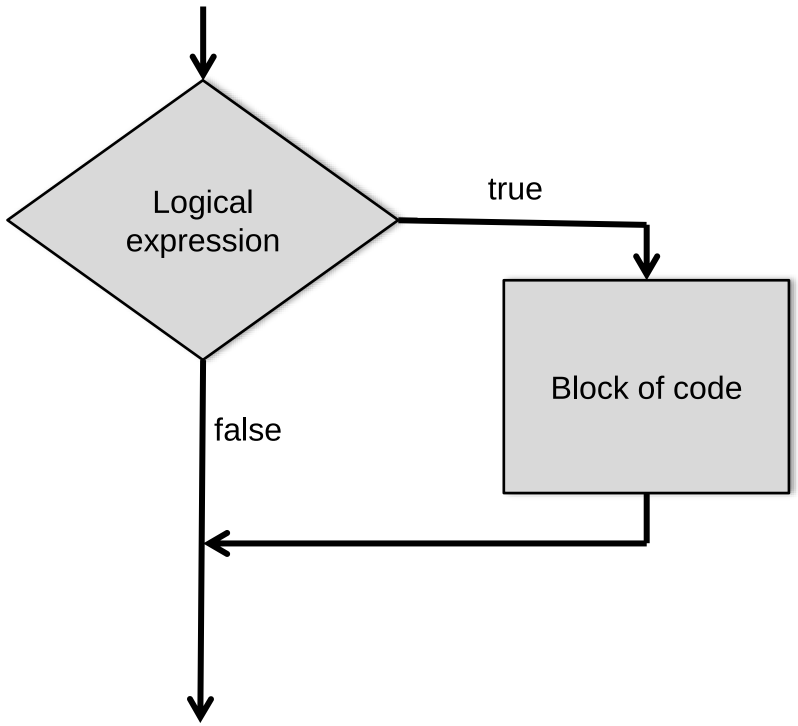
THE IF STATEMENT

- **Programming style suggestions:**
  - The block of code should be indented 3-4 spaces to aid program readability
  - If the block of code is only <u>one</u> line long, we can omit the curly brackets { } and shorten the length of the program

- **Never put a semi-colon directly after the Boolean expression in an if statement**
  - The empty statement between) and ; will be selectively executed based on the logical expression value
  - The block of code directly below if statement will **always** be executed, which is probably not what you intended

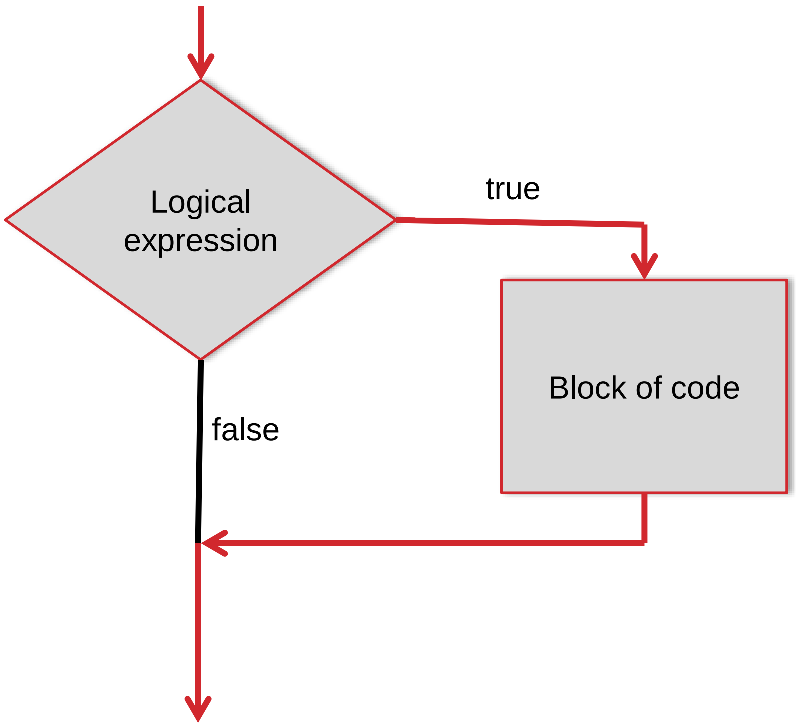
THE IF STATEMENT

- **We can visualize the program’s if statement decision process using a “flow chart” diagram**



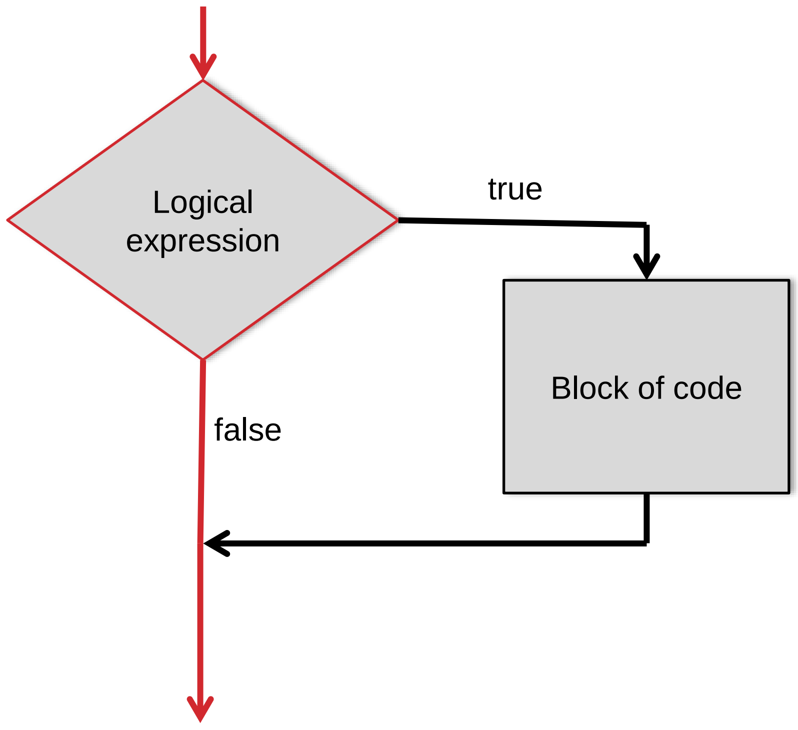
THE IF STATEMENT

- **If the logical expression is true, we take one path through the diagram (executing the block of code)**



THE IF STATEMENT

- **If the logical expression is false, we take a different path through the diagram (skipping over the block of code)**



THE IF STATEMENT

```
// Simple if statement
int a, b;
cin >> a >> b;
if (a < b)
{
   cout << “A is smaller than B\n”;
}
```

- Depending on what data values the user enters, the cout statement will executed or skipped

THE IF STATEMENT

```
// One line block of code
int a, b;
cin >> a >> b;
if (a == b)
   cout << “A is equal to B\n”;
```

- This is the same if statement as the previous example but we removed the curly brackets to shorten the program

THE IF STATEMENT

```
// Block of code that never executes
if (1 == 2)
{
   cout << “This code will never execute\n”;
}

// Block of code that always executes
if (1 < 2)
{
   cout << “This code will always execute\n”;
}
```

THE IF-ELSE STATEMENT

- **Sometimes we need to handle two alternatives in our code**
- **The C++ syntax of the if-else statement is:**

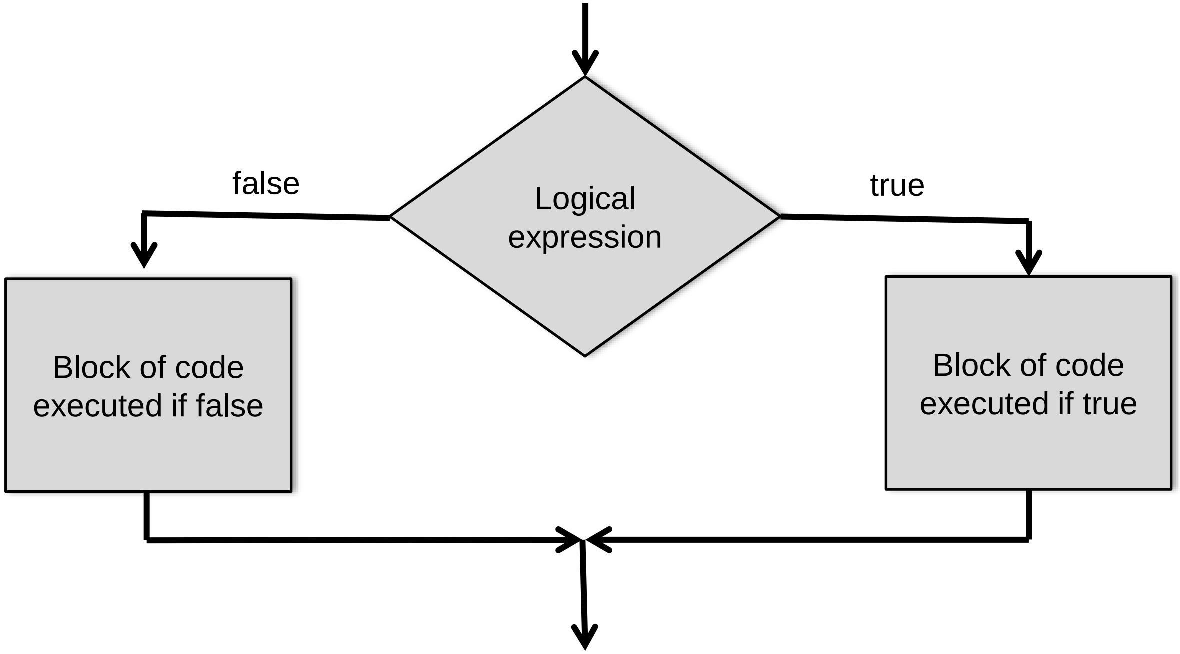
```
if ( logical expression )
{
   // Block of code to execute if expression is true
}
else
{
   // Block of code to execute if expression is false
}
```

THE IF-ELSE STATEMENT

- **Programming style suggestions:**
  - Type the “if line” and the “else line” and the { } brackets so they are vertically aligned with each other
  - Do not put a semi-colon after the “if line” or the “else line” or you will get very strange run time errors
  - The two blocks of code should be indented 3-4 spaces to aid program readability
  - If either block of code is only one line long, we can omit the curly brackets { } and shorten the length of the program

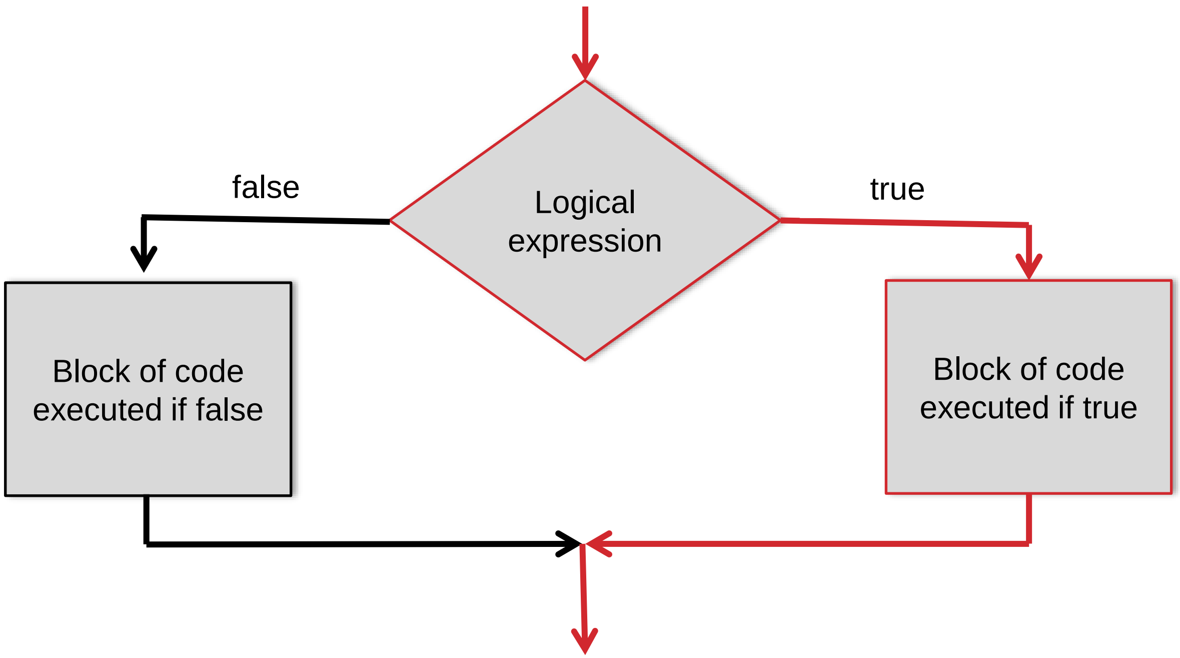
THE IF-ELSE STATEMENT

- **We can visualize the program's if-else statement decision process using a "flow chart" diagram**



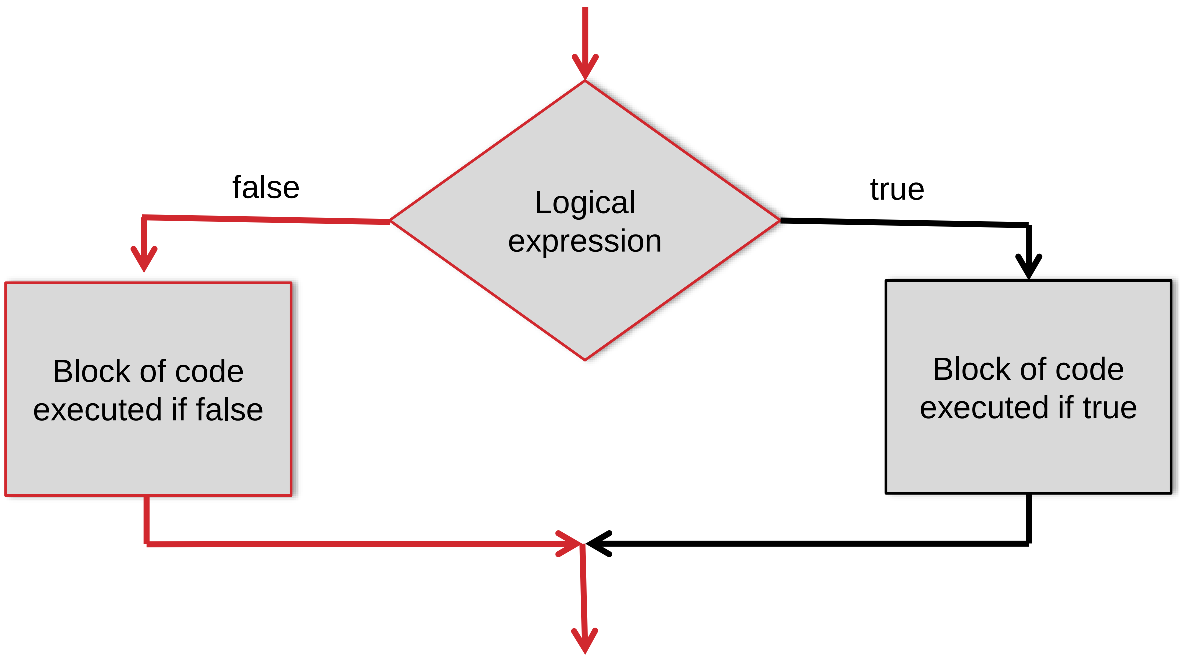
THE IF-ELSE STATEMENT

- **If the logical expression is true, we take one path through the diagram (executing one block of code)**



THE IF-ELSE STATEMENT

- **If the logical expression is false, we take one path through the diagram (executing the other block of code)**



THE IF-ELSE STATEMENT

```
// Simple if-else example
if ((a > 0) && (b > 0))
{
   c = a / b;
   a = a - c;
}
else
{
   c = a * b;
   a = b + c;
}
```

THE IF-ELSE STATEMENT

```
// Ugly if-else example
if (a < b) {
c = a * 3;
a = b - c; } else
a = c + 5;
```

- **This code is technically correct, but it is difficult for humans to read and understand the intended logic**

THE IF-ELSE STATEMENT

```
// Pretty if-else example
if (a < b)
{
   c = a * 3;
   a = b - c;
}
else
   a = c + 5;
```

Notice that the else part is only one line long so we omitted the curly brackets

- **This is the same portion of code with proper indentation so it is much easier for humans to read and understand**

GRADE CALCULATION EXAMPLE

- **How can we convert test scores to letter grades?**
  - We must read test scores with values between 0..100
  - We want to output corresponding A,B,C,D,F letter grades

- **To find the letter grade, we need a series of if statements**
  - If score is between 90..100 output A
  - If score is between 80..89 output B
  - If score is between 70..79 output C
  - If score is between 60..69 output D
  - If score is between 0..59 output F

GRADE CALCULATION EXAMPLE

- **It is very important to develop and test programs incrementally, just a few lines at a time**
  - Start by writing comments that describe the steps you want the program to take
  - Then add some code under each comment that implements that part of the program
  - Then compile and run the partial program to make sure there are no syntax errors, and that the part you have implemented is working correctly
  - Continue adding small pieces of code, compiling and testing the program until it is complete

GRADE CALCULATION EXAMPLE

```
// Program to convert test scores into letter grades
#include <iostream>
using namespace std;
int main()
{
   // Local variable declarations
   // Read test score
   // Calculate letter grade
   // Print output
   return 0;
}
```

The first step is to write comments in the main program to explain our approach

This will compile and run but not do anything

GRADE CALCULATION EXAMPLE

```
// Program to convert test scores into letter grades
#include <iostream>
using namespace std;
int main()
{
   // Local variable declarations
   float Score = 0;
   char Grade = '?' ;
   // Read test score
   cout << "Enter test score: ";
   cin >> Score;
   cout >> "Score: " << Score << endl;
   …
```

Next, we_add code to the main program to get the input test score

This will compile and run but only read and print the input test score

GRADE CALCULATION EXAMPLE

```
// Local variable declarations
float Score = 0;
char Grade = '?' ;
// Read test score
cout << “Enter test score: “;
cin >> Score;
cout >> “Score: ” << Score << endl;
// Calculate letter grade
if ((Score >= 90) && (Score <= 100))
   Grade = 'A';
// Print output
cout << “Grade: ” << Grade << endl;
…
```

Next, we add more code calculate one letter grade and then print output

This will compile and run but it will only calculate A grades correctly

GRADE CALCULATION EXAMPLE

```
…
// Calculate letter grade
if ((Score >= 90) && (Score <= 100))
   Grade = 'A';
if ((Score >= 80) && (Score < 90))
   Grade = 'B';
if ((Score >= 70) && (Score < 80))
   Grade = 'C';
if ((Score >= 60) && (Score < 70))
   Grade = 'D';
if ((Score >= 0) && (Score < 60))
   Grade = 'F';
…
```

Finally, we add more code to calculate the remaining letter grades

This will compile and run and hopefully calculate all grades

GRADE CALCULATION EXAMPLE

- **We should start testing with "expected" input values**
  - Try test scores that we know are in the middle of the A,B,C,D,F letter ranges (e.g. 95,85,75,65,55)
  - Try input values that are "on the border" of the letter grade ranges to make sure we have our ">=" and ">" conditions right (e.g. 79,80,81)

- **We should then test "unexpected" input values**
  - Try entering test values that are outside the 0..100 range to see what the program will output
  - Finally, see what happens if the user enters something other than an integer test score (e.g. 3.14159, "hello")

SUMMARY

- In this section we have studied the syntax and use of the C++ if statement and the if-else statement
- We have also seen how flow chart diagrams can be used to visualize different execution paths in a program
- Finally, we showed how if statements can be used to implement a simple grade calculation program

CONDITIONAL STATEMENTS

PART 3

NESTED IF STATEMENTS

NESTED IF STATEMENTS

- **We can have two or more if statements inside each other to check multiple conditions**
  - These are called nested if statements

- **Use indentation to reflect nesting and aid readability**
  - Typically indent 3-4 spaces or one tab per nesting level

- **Need to be careful when matching up { } brackets**
  - This way you can decipher the nesting of conditions

NESTED IF STATEMENTS

```
if ( logical expression1 )
{
   if ( logical expression2 )
   {
      // Statements to execute if expressions1 and expression2 are true
   }
   else
   {
      // Statements to execute if expression1 true and expression2 false
   }
}
else
{
   // Statements to execute if expression1 false
}
```

NESTED IF STATEMENTS

```
// Simple nested if example
cin >> a >> b;
if (a < b)
{
   cout << "A is smaller than B\n";
   if ((a > 0) && (b > 0))
      cout << "A and B are both positive\n";
   else
      cout << "A or B or both are negative\n";
}
```

NESTED IF STATEMENTS

```
// Ugly nested if example
if (a > 0) {
if (b < 0) {
   a = 3 * b;
   c = a + b; } }
else {
   a = 2 * a;
   c = b / a; }
```

It is hard to see what if statement the else code goes with

NESTED IF STATEMENTS

```
// Pretty nested if example
if (a > 0)
{
   if (b < 0)
   {
      a = 3 * b;
      c = a + b;
   }
}
else
{
   a = 2 * a;
   c = b / a;
}
```

Now we can see the else goes with the first if statement

NESTED IF STATEMENTS

- We can use nested if statements to calculate grades with fewer comparison operations then the previous example
- The key is to make use of what we know is true when we go into the "else" block of code and not test this again

```
if (Score >= 90)
   Grade = 'A';
else
{
   …
}
```

We know Score < 90 so we do not need to test for this again

NESTED IF STATEMENTS

```
if (Score >= 90)
   Grade = 'A';
else
{
    if (Score >= 80)
       Grade = ‘B';
   else
   {
       …
   }
}
```

We also know Score < 90 so the score is in the B range (arrow pointing to `if (Score >= 80)`)

We know Score < 80 so we do not need to test for this again (arrow pointing to `…`)

NESTED IF STATEMENTS

```
if (Score >= 90)
   Grade = 'A';
else if (Score >= 80)
   Grade = 'B';
else if (Score >= 70)
   Grade = 'C';
else if (Score >= 60)
   Grade = 'D';
else if (Score >= 0)
   Grade = 'F';
```

Since each else block is only one line long we can omit the curly brackets to save space

We can also line up all of the "else if" statements with the original if statement

BOOLEAN VARIABLES

- **In C++ we can store true/false values in Boolean variables**
- **The constants true and false can be used to initialize bool variables**
  - bool Done = true;
  - bool Quit = false;
- **Boolean expressions can also be used to initialize bool variables**
  - int a = 2, b = 3;
  - bool Positive = (a >= 0);
  - bool Negative = (b < 0);

BOOLEAN VARIABLES

- **Boolean variables and true/false constants can also be used in logical expressions**
  - (Done == true) is true
  - (Quit != true) is true
  - (Done == Quit) is false
  - (true == Positive) is true
  - ((a < b) == false) is false
  - (Negative) is false

BOOLEAN VARIABLES

- **Internally C++ stores Boolean variables as integers**
  - 0 is normally used for false
  - 1 is normally used for true
  - Any value not equal to 0 is considered true

- **It is considered "bad programming style" to use integers instead of the true/false keywords**
  - bool Good = 0;
  - bool Bad = 1;
  - bool Ugly = 2;

BOOLEAN VARIABLES

- **Integers are used when writing Boolean values**
  - cout << Good will print 0
  - cout << Bad will print 1
  - cout << Ugly will also print 1

- **Integers are also used when reading Boolean values**
  - cin >> Good;
  - Entering 0 sets Good variable to false
  - Entering any value >= 1 sets Good variable to true
  - Entering value < 0 also sets Good variable to true
  - Entering “true” or “false” will <u>not</u> work

BOOLEAN VARIABLES

- **Boolean variables are often used for status flags**
  - Set status flag to initial value
  - Test to see if certain condition occurs
  - Update status flag when necessary

```
bool Positive = true;
if (a < 0) Positive = false;
if (b < 0) Positive = false;
if (c < 0) Positive = false;
```

PRIME NUMBER EXAMPLE

- **How can we test a number to see if it is prime?**
  - We are given numerical values between 1..100
  - We need to see if it has any factors besides 1 and itself
  - If no factors found then number is prime

- **We need some nested if statements**
  - Test if input number is between 1..100
  - If so, then test if 2,3,5,7 are factors of input number
  - Then print out “prime” or “not prime”

PRIME NUMBER EXAMPLE

- **How can we test a if F is a factor of N?**
  - By definition “A factor of N is an integer F that may be multiplied by some other integer to produce N”
  - N = F * V for some integer V
  - N / F = V with no remainder
  - (F * (N / F) == N) true if F a factor
  - (N % F == 0) true if F a factor

- **To be a prime factor, F can not equal N**
  - ((N != F) && (N % F == 0))

PRIME NUMBER EXAMPLE

```
// Check for prime numbers using a factoring approach
#include <iostream>
using namespace std;

int main()
{
   // Local variable declarations
   // Read input parameters
   // Check input is valid
   // Check if number is prime
   // Print output
   return 0;
}
```

For the first version of program we just write comments in the main program to explain our approach

PRIME NUMBER EXAMPLE

```
// Check for prime numbers using a factoring approach
#include <iostream>
using namespace std;

int main()
{
 // Local variable declarations
 int Number = 0;
 bool Prime = true;
 // Read input parameters
 cout << "Enter a number [1..100]:";
 cin >> Number;
```

For the second version of program we initialize variables and read the input value from user

PRIME NUMBER EXAMPLE

```
…
  cout << “Enter a number [1..100]:”;
  cin >> Number;
  // Check input is valid
  if ((Number < 1) || (Number > 100))
     cout << “Error: Number is out of range\n”;
  else
   {
     // Check if number is prime
     // Print output
  }
```

For the next version of the program we add code to verify the range of input value

PRIME NUMBER EXAMPLE

```
…
  cout << “Enter a number [1..100]:”;
  cin >> Number;
  // Check input is valid
  if ((Number >= 1) && (Number <= 100))
  {
     // Check if number is prime
     // Print output
  }
  else
     cout << “Error: Number is out of range\n”;
```

For the next version of the program we add code to verify the range of input value

PRIME NUMBER EXAMPLE

…

```
// Check if number is prime
if (Number == 1) Prime = false;
if ((Number != 2) && (Number % 2 == 0)) Prime = false;
if ((Number != 3) && (Number % 3 == 0)) Prime = false;
if ((Number != 5) && (Number % 5 == 0)) Prime = false;
if ((Number != 7) && (Number % 7 == 0)) Prime = false;
// Print output
 if (Prime)
   cout << “Number “ << Number << “ IS prime\n”;
else
   cout << “Number “ << Number << “ is NOT prime\n”;
```

In the final version we finish the prime number calculation and print the output

PRIME NUMBER EXAMPLE

- **How should we test the prime number program?**
  - Test the range checking code by entering values “on the border” of the input range (e.g. 0,1,2 and 99,100,101)
  - Test program with several values we know <u>are</u> prime
  - Test program with several values we know are <u>not</u> prime
  - To be really compulsive we could test all values between 1..100 and compare to known prime numbers

- **What is wrong with this program?**
  - It only works for inputs between 1..100
  - It will not “scale up” easily if we extend this input range

SUMMARY

- In this section we showed how if statements and if-else statements can be nested inside each other to create more complex paths through a program
- We also showed how proper indenting is important to read and understand programs with nested if statements
- We have seen how Boolean variables can be used to store true/false values in a program
- Finally, we used an incremental approach to create a program for checking the factors of input numbers to see if they are prime or not

CONDITIONAL STATEMENTS

PART 4

SWITCH STATEMENTS

SWITCH STATEMENTS

- **The switch statement is convenient for handling multiple branches based on the value of one decision variable**
  - The program looks at the value of the decision variable
  - The program jumps directly to the matching case label
  - The statements following the case label are executed
- **Special features of the switch statement:**
  - The “break” command at the end of a block of statements will make the program jump to the end of the switch
  - The program executes the statements after the “default” label if no other cases match the decision variable

SWITCH STATEMENTS

```
switch ( decision variable )
{
   case value1 :
      // Statements to execute if variable equals value1
      break;
   case value2:
      // Statements to execute if variable equals value2
      break;
   ...
   default:
      // Statements to execute if variable not equal to any value
}
```

SWITCH STATEMENTS

```
int Age = 0;
cin >> Age;
switch (Age)
{
  case 0:
    cout << “Stop being such a baby” << endl;
    break;
  case 7:
    cout << “Are you going to first grade now?” << endl;
    break;
```

The program will execute this code only if Age == 0

SWITCH STATEMENTS

```
  case 21:
    cout << "Lets go for a drink" << endl;
    break;
  case 42:
    cout << "This is the ultimate age" << endl;
    break;
  default:
    cout << "Your age is boring" << endl;
}
```

SWITCH STATEMENTS

```
char Choice = ' ';
cin >> Choice;
switch (Choice)
{
  case 'd':  case 'D':
    cout << "Deposit money in bank" << endl;
    break;
  case 'w':  case 'W':
    cout << "Withdraw money from bank" << endl;
    break;
```

The program will execute this code only if Choice is 'd' or 'D'

SWITCH STATEMENTS

```
  case 't':  case 'T':
    cout << "Transfer money between accounts" << endl;
    break;
  case 'q':  case 'Q':
    cout << "Quit banking program" << endl;
    break;
  default:
    cout << "Invalid command" << endl;
}
```

SWITCH STATEMENTS

- **The main advantage of switch statement over a sequence of if-else statements is that it is much faster**
  - Jumping to blocks of code is based on a lookup table instead of a sequence of variable comparisons

- **The main disadvantage of switch statements is that the decision variable must be an integer or a character**
  - We can not use a switch with a float or string decision variable or with complex logical expressions

MENU EXAMPLE

- **How can we create a user interface for banking?**
  - Assume user selects commands from a menu
  - We need to see read and process user commands
- **We can use a switch statements to handle menu**
  - Ask user for numerical code for user command
  - Jump to the code to process that banking operation
  - Repeat until the user quits the application

MENU EXAMPLE

```
// Simulate bank deposits and withdrawals
#include <iostream>
using namespace std;
int main()
{
   // Local variable declarations
   // Print command prompt
   // Read user input
   // Handle banking command
   return 0;
}
```

For the first version of program we just write comments in the main program to explain our approach

MENU EXAMPLE

…

```
// Local variable declarations
int Command = 0;
// Print command prompt
cout << "Enter command number:\n";
// Read user input
cin >> Command;
```

…

For the next version of program we add the code to read the user command

MENU EXAMPLE

// Handle banking command

```
switch (Command)
{
case 0: // Quit code
   break;
case 1: // Deposit code
   break;
case 2: // Withdraw code
   break;
case 3: // Print balance code
   break;
}
```

Then we add the skeleton of the switch statement to handle the user command

MENU EXAMPLE

```
// Simulate bank deposits and withdrawals
#include <iostream>
using namespace std;
int main()
{
  // Local variable declarations
  int Command = 0;
  int Money = 0;
  int Balance = 100;
  // Print command prompt
  cout << “Enter command number:\n”
        << “   0 - quit\n”
        << “   1 - deposit money\n”
        << “   2 - withdraw money\n”
        << “   3 - print balance\n”;
```

In the final version add bank account variables and add code to perform banking operations

MENU EXAMPLE

```
// Read and handle banking commands
cin >> Command;
switch (Command)
{
case 0: // Quit code
   cout << “See you later!” << endl;
   break;
case 1: // Deposit code
   cout << “Enter deposit amount: “;
   cin >> Money;
   Balance = Balance + Money;
   break;
```

MENU EXAMPLE

```
case 2: // Withdraw code
  cout << “Enter withdraw amount: “;
  cin >> Money;
  Balance = Balance - Money;
  break;
case 3: // Print balance code
  cout << “Current balance = “ << Balance << endl;
  break;
default: // Handle other values
  cout << “Ooops try again” << endl;
  break;
 }
// Print final balance
cout << “Final balance = “ << Balance << endl;
}
```

MENU EXAMPLE

- **First, we should test program with "normal" inputs**
  - Try entering all valid menu commands
  - Try variety of deposit/withdraw amounts
- **Then, we should test with "abnormal" inputs**
  - What happens if we enter an invalid menu command?
  - What happens if we enter a negative input value?
  - What happens if the withdraw amount is larger then the account balance?
- **If we find problems, we should fix them or document them**

IMPROVED MENU EXAMPLE

- **To improve the menu, we can use letters that match the commands d=deposit, w=withdrawal instead of numbers**
  - Print letter based command menu
  - Read in letters from user
  - Convert switch cases to letters

- **To avoid negative balances, we must check to see if there is enough money in account before doing the withdrawal**
  - This requires an if statement inside the switch
  - Only do the withdrawal if the amount is valid
  - Print error message if withdrawal amount is invalid

IMPROVED MENU EXAMPLE

```
// Print command prompt
cout << “Enter command character:\n”
        << “   q / Q - quit\n”
        << “   d / D - deposit money\n”
        << “   w / W - withdraw money\n”
        << “   p / P - print balance\n”;
```

```
// Read user input
char Command = ‘ ‘;
cin >> Command;
```

Read single letter for user command

IMPROVED MENU EXAMPLE

// Handle banking command

```
switch (Command)
{
case ‘q’: case ‘Q’:   // Quit code
   break;
case ‘d’: case ‘D’:   // Deposit code
   break;
case ‘w’: case ‘W’:   // Withdraw code
   break;
case ‘p’: case ‘P’:    // Print balance code
   break;
}
```

Our new switch statement will use single character to select a command

IMPROVED MENU EXAMPLE

```
case ‘w’: case ‘W’:     // Withdraw code
   cout << “Enter withdraw amount: “;
   cin >> Money;
   if ((Money <= Balance) && (Money > 0))
      Balance = Balance - Money;
   else
      cout << “Can not withdraw money\n”;
   break;
```

This if statement does error checking before withdrawing the money

SOFTWARE ENGINEERING TIPS

- **There are many ways to write conditional code**
  - Your task is to find the simplest correct code for the task
- **Make your code easy to read and understand**
  - Indent your program to reflect the nesting of blocks of code
- **Develop your program incrementally**
  - Compile and run your code frequently
- **Anticipate potential user input errors**
  - Check for normal and abnormal input values

SOFTWARE ENGINEERING TIPS

- **Common programming mistakes**
  - Missing or unmatched ( ) brackets in logical expressions
  - Missing or unmatched { } brackets in conditional statement
  - Missing break statement at bottom of switch cases
  - Never use & instead of && in logical expressions
  - Never use | instead of || in logical expressions
  - Never use = instead of == in logical expressions
  - Never use “;” directly after logical expression

SUMMARY

- **In this section we have studied the syntax and use of the C++ switch statement**
- **We also showed an example where a switch statement was used to create a menu-based banking program**
- **Finally, have discussed several software engineering tips for creating and debugging conditional programs**