# Branching Statement And Logical Operators:

# If statement:

When a C++ program must choose whether to take a particular action, you usually implement the choice with an if statement. The if comes in two forms: if and if else.

The if statement directs a program to execute a statement or statement block if a test condition is true and to skip that statement or block if the condition is false.

Syntax:

A true test-condition causes the program to execute statement ,which can be a single statement or a block. A false test-condition causes the program to skip statement.

See the program – CountSpace.cpp

## The If-Else statement:

If else statement lets a program decide which of two statements or blocks is executed.

General Form:

```
if (test-condition)
        statement1
else
        statement2
```

If test-condition is true, or nonzero, the program executes statement1 and skips over statement2. Otherwise, when test-condition is false, or zero, the program skips statement1 and executes statement2 instead.

See the program – IfElse1.cpp

# The *if else if else* Construction:

See the program – GuessTheNumberElse.cpp

You can also use nested else if which is an example where you have to use if else condition inside an if else condition and so on to make it more complex.

# Logical Expressions:

These operators are :

1. Logical OR, written as ||.
2. Logical And, written as &&.
3. Logical Not, written as !.

Logical values require two expressions.

## Logical OR, operator: ||

In two expressions, if either one or two of them is true, or nonzero or the resulting expression has the value true. Otherwise the expression has the value false.

```
5 == 5 || 5 == 9   // true because first expression is true

5 > 3 || 5 > 10    // true because first expression is true
```

5 > 8 || 5 < 10  // true because second expression is true

5 < 8 || 5 > 2  // true because both expressions are true

5 > 8 || 5 < 2  // false because both expressions are false.

Because the || has a lower precedence than the relational operators, you don't need to use parentheses in these expressions

C++ provides that the || operator is a sequence point. That is, any value changes indicated on the left side take place before the right side is evaluated.

## Logical AND, operator: &&

The resulting expression has the value true only if both of the original expressions are true.

E.g.

5 == 5 && 4 == 4  // true because both expressions are true

5 == 3 && 4 == 4  // false because first expression is false

5 > 3 && 5 > 10  // false because second expression is false

5 > 8 && 5 < 10  // false because first expression is false

5 < 8 && 5 > 2  // true because both expressions are true

5 > 8 && 5 < 2  // false because both expressions are false

Because the && has a lower precedence than the relational operators, you don't need to use parentheses in these expressions. Like the || operator, the && operator acts as a sequence point, so the left side is evaluated, and any side effects are carried out before the right side is evaluated. If the left side is false, the whole logical expression must be false, so C++ doesn't bother evaluating the right side in that case.

## Range using *AND*:

e.g.

if (age > 18 && age < 50) // ok

But don't do this,

if (18 < age < 50)

If you do this mistake the compiler won't catch it as an error because it is still a valid C++ syntax. The < operator associates from left to right, so the previous expression means:

if ((18 < age) < 50)

But 18 < age is either True (or 1) or else false (or 0). In either case, the expression 18 < age is less than 50, so the entire test is always true.

## Logical NOT, operator: !

The ! operator negates, or reverses the truth value of, the expression that follows it. That is, if expression is true, then ! expression is false—and vice versa. More precisely, if expression is true, or nonzero, then !expression is false. Incidentally, many people call the exclamation point bang, making !x "bang-ex" and !!x "bang-bang-ex."

But the ! operator can be useful with functions that return true/false values or values that can be interpreted that way.

## Logical Operator Facts:

The C++ logical OR and logical AND operators have a lower precedence than relational operators. This means that an expression such as this:

x > 5 && x < 10

is interpreted this way:

(x > 5) && (x < 10)

The ! operator, on the other hand, has a higher precedence than any of the relational or arithmetic operators. Therefore, to negate an expression, you should enclose the expression in parentheses, like this:

```
!(x > 5)     // is it false that x is greater than 5
!x > 5       // is !x greater than 5
```

Incidentally, the second expression here is always false because !x can have only the values true or false, which get converted to 1 or 0.

The logical AND operator has a higher precedence than the logical OR operator. Thus this expression:

age > 30 && age < 45 || weight > 300

means this:

(age > 30 && age < 45) || weight > 300

## Alternative Representations:

Not all keyboards provide all the symbols used for the logical operators, so the C++ Standard provides alternative representations.

| Operator | Alternative Representation |
|---|---|
| && | and |
| \|\| | or |
| ! | not |

The identifiers and, or, and not are C++ reserved words, meaning that you can't use them as names for variables and so on. They are not considered keywords because they are alternative representations of existing language features. Incidentally, these are not reserved words in C, but a C program can use them as operators, provided that the program includes the iso646.h header file. C++ does not require using a header file.

## The cctype Library of character functions:

C++ has inherited from C a handy package of character-related functions, prototyped in the cctype header file (ctype.h, in the older style),that simplify such tasks as determining whether a character is an uppercase letter or a digit or punctuation. For example, the isalpha(ch) function returns a nonzero value if ch is a letter and a zero value otherwise. Similarly, the ispunct(ch) function returns a true value only if ch is a punctuation character, such as a comma or period.(These functions have return type int rather than bool, but the usual bool conversions allow you to treat them as type bool.)

Using these functions is more convenient than using the AND and OR operators.

E.g.

```
if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'))
```

Compare that to using isalpha():

```
if (isalpha(ch))
```

You can use,

isdigits() // Test for digit characters

isspace() // Test for whitespaces characters

See the program – cctype.cpp

## The cctype character functions:

| Function Name | Return Value |
|---|---|
| isalnum() | This function returns true if the argument is alphanumeric (that is, a letter or a digit). |
| isalpha() | This function returns true if the argument is alphabetic. |
| isblank() | This function returns true if the argument is a space or a horizontal tab. |
| iscntrl() | This function returns true if the argument is a control character. |
| isdigit() | This function returns true if the argument is a decimal digit (0–9). |
| isgraph() | This function returns true if the argument is any printing character other than a space |
| islower() | This function returns true if the argument is a lowercase letter. |
| isprint() | This function returns true if the argument is any printing character, including a space. |
| ispunct() | This function returns true if the argument is a punctuation character. |
| isspace() | This function returns true if the argument is a standard whitespace character (that is, a space, formfeed, newline, carriage return, horizontal tab, vertical tab). |
| isupper() | This function returns true if the argument is an uppercase letter. |
| isxdigit() | This function returns true if the argument is a hexadecimal digit character (that is, 0–9, a–f, or A–F). |
| tolower() | If the argument is an uppercase character, tolower() returns the lowercase version of that character; otherwise, it returns the argument unaltered. |
| toupper() | If the argument is a lowercase character, toupper() returns the uppercase version of that character; otherwise, it returns the argument unaltered. |

# The ?: operator:

C++ has an operator that can often be used instead of the if else statement. This operator is called the conditional operator, written ?:, and, for you trivia buffs, it is the only C++ operator that requires three operands.

General Form:

expression1 ? expression2 : expression3

If expression1 is true, then the value of the whole conditional expression is the value of expression2. Otherwise, the value of the whole expression is the value of expression3.

E.g.

5 > 3 ? 10 : 12  // 5 > 3 is true, so expression value is 10

3 == 9? 25 : 18  // 3 == 9 is false, so expression value is 18

See the program – CndtlOprtr.cpp

```
int c = a > b ? a : b;
```

It produces the same result as the following statements:

```
int c;
if (a > b)
        c = a;
else
        c = b;
```

.The conditional operator's concise form, unusual syntax, and overall weird appearance make it a great favorite among programmers who appreciate those

qualities. One favorite trick for the reprehensible goal of concealing the purpose of code is to nest conditional expressions within one another, as the following mild example shows:

```
const char x[2] [20] = {"Jason ","at your service\n"};

const char * y = "Quillstone ";

for (int i = 0; i < 3; i++)

        cout << ((i < 2)? !i ? x [i] : y : x[1]);
```

This is merely an obscure (but, by no means maximally obscure) way to print the three strings in the following order:

Jason Quillstone at your service

In terms of readability, the conditional operator is best suited for simple relationships and simple expression values.

If the code becomes more involved, it can probably be expressed more clearly as an if else statement.

# The *switch* statement:

Suppose you create a screen menu that asks the user to select one of five choices—for example, Cheap, Moderate, Expensive, Extravagant, and Excessive. You can extend an if else if else sequence to handle five alternatives, but the C++ switch statement more easily handles selecting a choice from an extended list.

General Form:

```
switch ( integer-expression ) {
case label1 : statement(s)
case label2 : statement(s)
...
default     : statement(s)
}
```

.The value integer-expression ,as the name suggests, must be an expression that reduces to an integer value. Also each label must be an integer constant expression. Most often, labels are simple int or char constants, such as 1 or 'q', or enumerators. If integer-expression doesn't match any of the labels, the program jumps to the line labeled default. The default label is optional. If you omit it and there is no match, the program jumps to the next statement following the switch.

Each C++ case label functions only as a line label, not as a boundary between choices. That is, after a program jumps to a particular line in a switch, it then sequentially executes all the statements following that line in the switch unless you explicitly direct it otherwise. Execution does not automatically stop at the next case. To make execution stop at the end of a particular group of statements, you must use the break statement. This causes execution to jump to the statement following the switch.

See the program – Switch1.cpp

## Using Enumerators as Labels:

In general, cin doesn't recognize enumerated types (it can't know how you will define them),so the program reads the choice as an int. When the switch statement compares the int value to an enumerator case label, it promotes the enumerator to int. Also the enumerators are promoted to type int in the while loop test condition.

See the program – SwitchEnum.cpp

**If you can use either an if else if sequence or a switch statement, the usual practice is to use switch if you have three or more alternatives**.

## The *break* and *continue* statements:

The break and continue statements enable a program to skip over parts of the code. You can use the break statement in a switch statement and in any of the loops. It causes program execution to pass to the next statement following the switch or the loop. The continue statement is used in loops and causes a program to skip the rest of the body of the loop and then start a new loop cycle.

See the program – BrkCnt1.cpp

## Simple File Input/Output:

Sometimes keyboard input is not the best choice. For example, suppose you've written a program to analyze stocks, and you've downloaded a file of 1,000 stock prices. It would be far more convenient to have the program read the file directly than to hand-enter all the values. Similarly, it can be convenient to have a program write output to a file so that you have a permanent record of the results. Fortunately, C++ makes it simple to transfer the skills you've acquired for keyboard input and display output (collectively termed console I/O) to file input and output (file I/O).

# Text I/O and Text Files:

### *Writing to a Text File*:

For file output, C++ uses analogs to cout.

So to prepare for file output, let's review some basic facts about using cout for console output:

1. You must include the iostream header file.
2. The iostream header file defines an ostream class for handling output.
3. The iostream header file declares an ostream variable, or object, called cout.
4. You must account for the std namespace; for example, you can use the using directive or the std:: prefix for elements such as cout and endl.
5. You can use cout with the << operator to read a variety of data types.

File output parallels this very closely:

1. You must include the fstream header file.
2. The fstream header file defines an ofstream class for handling output.
3. You need to declare one or more ofstream variables, or objects, which you can name as you please, as long as you respect the usual naming conventions.
4. You must account for the std namespace; for example, you can use the using directive or the std:: prefix for elements such as ofstream.
5. You need to associate a specific ofstream object with a specific file; one way to do so is to use the open() method.
6. When you're finished with a file, you should use the close() method to close the file.
7. You can use an ofstream object with the << operator to output a variety of data types.



## Declaring Objects:

```
ofstream outFile;          // outFile an ofstream object
ofstream fout;             // fout an ofstream object
```

## Associate the objects with particular files:

```
outFile.open("fish.txt");  // outFile used to write to the fish.txt file
char filename[50];
cin >> filename;           // user specifies a name
```

```
fout.open(filename);       // fout used to read specified file
```

Note that the open() method requires a C-style string as its argument. This can be a literal string or a string stored in an array.

## Using the objects:

```
double wt = 125.8;
outFile << wt;        // write a number to fish.txt
char line[81] = "Objects are closer than they appear.";
fout << line << endl;   // write a line of text
```

The important point is that after you've declared an ofstream object and associated it with a file, you use it exactly as you would use cout. All the operations and methods available to cout, such as <<,endl, and setf(), are also available to ofstream objects.

## Main steps for using file output:

1. Include the fstream header file.
2. Create an ofstream object.
3. Associate the ofstream object with a file.
4. Use the ofstream object in the same manner you would use cout.

See the program – OutputFile.cpp

When the program is done using a file, it should close the connection:

```
outFile.close();
```

Notice that the close() method doesn't require a filename. That's because outFile has already been associated with a particular file. If you forget to close a file, the program will close it automatically if the program terminates normally.

## Caution:

**When you open an existing file for output, by default it is truncated to a length of zero bytes, so the contents are lost**.

## Reading from a Text File:

For File Input:

1. You must include the fstream header file.
2. The fstream header file defines an ifstream class for handling input.
3. You need to declare one or more ifstream variables, or objects, which you can name as you please, as long as you respect the usual naming conventions.
4. You must account for the std namespace; for example, you can use the using directive or the std:: prefix for elements such as ifstream.
5. You need to associate a specific ifstream object with a specific file; one way to do so is to use the open() method.
6. When you're finished with a file, you should use the close() method to close the file.
7. You can use an ifstream object with the >> operator to read a variety of data types.
8. You can use an ifstream object with the get() method to read individual characters and with the getline() method to read a line of characters at a time.
9. You can use an ifstream object with methods such as eof() and fail() to monitor the success of an input attempt.
10. An ifstream object itself, when used as a test condition, is converted to the Boolean value true if the last read attempt succeeded and to false otherwise.

See the program – AvgTxtFile.cpp

# Caution:

A Windows text file uses the carriage return character followed by a linefeed character to terminate a line of text. (The usual C++ text mode translates this combination to newline character when reading a file and reverses the translation when writing to a file.) Some text editors, such as the Metrowerks CodeWarrior IDE editor, don't automatically add this combination to the final line. Therefore, if you use such an editor, you need to press the Enter key after typing the final text and before exiting the file.

# Thank You