Control Statements Cont.

Chapter 5, 6

If/Else, Loop

Combining Control Statements

```
if(          )
{
    while(       )
    {


    }
}
```

```
while(          )
{
    if(       )
    {


    }
}
```

- Control statements can be nested into other control statements to build more complex program paths
- Typical programs will have nested statements with a depth greater than 10

Guided Example 5.1

- Write a program that allows the user to select whether the program will calculate the area of a triangle or of a rectangle. At the end of the program prompt the user if they wish to perform another calculation

Absence of Curly Braces

- In the absence of curly braces for an if/else or loop construct, the next statement/construct, and only the next statement/construct shall be considered the body of the loop.

- Example

```
if(x < 10)
     cout << "x is less than 10" << endl;
cout << "this is outside the conditional" << endl;
```

Else If Statements

Consider the menu selection problem:

- User shall Select:
  - 1 for triangle
  - 2 for rectangle
- Automatically detect invalid selection

Else – If Statements

- We learned to write the statements as the following:

```
if(select == 1)
{
    // Compute triangle
}
else
{
    if(select == 2)
    {
        // Compute Rectangle
    }
    else
    {
        // Invalid Selection
    }
}
```

Utilizing the rule that we do not require to write curly braces for a single instruction/construct conditional we can simplify the structure

Else – If Statements

```
if(select == 1)
{
   // Compute triangle
}
else if(select == 2)
{
   // Compute rectangle
}
else
{
   // Invalid Selection
}
```

- Resulting structure is equivalent to the previous
- No change in the execution path compared to previous
- Why do this?
  - Easier to read

Consider the Following Problem

- Write a program that prints *success* if the variable *x* is: 10 < x < 20, else print *failure*.

```
int x;
cin >> x;
if( x < 20)
{
        if(x > 10)
                cout << "success" << endl;
        else
                cout << "failure" << endl;
}
else
   cout << "failure" << endl;
```

Take away

```
if( x < 20)
{
        if(x > 10)
                cout << "success" << endl;
        else
                cout << "failure" << endl;
}
else
   cout << "failure" << endl;
```

- Code required repeating in the else statement
- the path is complex to evaluate a simple condition
- Why cant I evaluate x < 20, x > 10 in the same condition?

Logical Operators

- Logical Operators allow for combining multiple boolean operands into a single logical expression.
- The Operators
  - And: &&
  - Or: ||
  - Not: !

- **Do not** use & or | for a logical operations; these are bitwise operations and it will compile.

The And Statement

- Language example:
  - Your mom says, “At the grocery store buy eggs and milk”
  - If you forget eggs or milk you did not follow the instruction (false)
  - If you got both you then you followed the instruction. (true)

bool eggs, milk;

C++ example: eggs && milk

Truth Table

Eggs	Milk	Output
False	False	False
False	True	False
True	False	False
True	True	True

The Or Statement

- Language example:
  - Your mom says, “At the grocery store buy eggs or milk”
  - If you forget both eggs and milk you did not follow the instruction (false)
  - If you got either eggs or milk you then you followed the instruction. (true)

bool eggs, milk;

C++ example: eggs || milk

Truth Table

Eggs	Milk	Output
False	False	False
False	True	True
True	False	True
True	True	True

The Not Statement

- Language example:
  - Your teacher says, “Don’t skip class”
  - If you skip then you didn’t follow your teachers wishes (false)
  - If you didn’t skip, then you followed then you followed your teachers wishes (true)

bool skip;

!skip

Truth Table

Skip	Output
False	True
True	False

Revisiting the Problem

- Write a program that prints *success* if the variable *x* is: 10 < x < 20, else print *failure*.

```
int x;
cin >> x;
if(10 < x  &&  x < 20)
   cout << "success" << endl;
else
   cout << "failure" << endl;
```

Order of Operations

Precedence	Operator	Associativity
Highest	! unary - ++ --	Right to Left
…	* / %	Left to Right
…	+ -	Left to Right
…	< <= > >=	Left to Right
…	== !=	Left to Right
…	&&	Left to Right
…	||	Left to Right
Lowest	= += -= *= /=	Right to Left

Examples of Logical Expressions

- i + 2 == k – 1
  - false
- 3 * i – j < 2
  - false
- i + 2 * j > k
  - true
- k + 3 <= - j + 3 * i
  - false
- ‘a’ + 1 == ‘b’
  - true
- key + 1 == ‘n’
  - true
- 25 >= x + 1.0
  - true

```
char key = 'm';
int i = 5, j = 7, k = 12;
double x = 22.5;
```

Example 5.2

- Write a program that allows a char to be input by the user and detects whether the char is lower case, upper case or not a letter

Example 5.2 Solution

```
char letter;
cout << "Enter a character: ";
cin >> letter;

if(letter >= 'a' && letter <= 'z')
	cout << "the character is lowercase" << endl;
else if(letter >= 'A' && letter <= 'Z')
	cout << "the character is uppercase" << endl;
else
	cout << "that ain't no letter i've seen" << endl;
```

Example 5.3

- Write a program that allows a char to be input by the user and detects whether the char is a letter or not a letter

Example 5.3 Solution

```
char letter;
cout << "Enter a character: ";
cin >> letter;

if(letter >= 'a' && letter <= 'z' || letter >= 'A' && letter <= 'Z')
   cout << "the character is a letter" << endl;
else
   cout << "that ain't no letter i've seen" << endl;
```

Unguided Example 5.4

- Write a program that allows the user to select whether the program will calculate the area of a triangle or of a rectangle. At the end of the program prompt the user if they wish to perform another calculation.

- For all prompts allow the user to make a selection with both upper or lower case letters.

For Loop

```
for(initialization; condition; increment)
{
   body of for loop
}
```

- For loops are a specialized loop for counting
- The initialization of variables, and counting is done in the loop header

For Loop Cont.

```
for(initialization; condition; increment)
{
   body of for loop
}
```

- Composed as three segments separated by semicolons
- Initialization
  - Used to initialize variables to be used in the loop, especially counting variables
- Condition
  - same as while loop
- increment
  - while any statement can be put here, reserve it exclusively for modification of the counting variable

Example of For Loop

```
int i;
for(i = 0; i < 5; i++)
{
   cout << i << endl;
}
```

Output:

0, 1, 2, 3, 4

- Notice simplification of counting

- Note: all **For** loops can be implemented as **While** loops and vice versa

Example of For Loop

```
int i;

for(i = 0;   i < 5;   i++)
{
3 cout << i << endl;

}
```

1 2 4

Sequence:

1. Initialization (occurs only once on entrance)
2. Evaluate condition, execute body of loop if true
3. Execute body of the loop
4. Execute counter to prepare for next iteration

Guided Example 5.5

- Write a program using a **For** loop to calculate the result of a number raised to a power. The user will input the base and the exponent. For instance, 3^4 will result in 81.

Unguided Example 5.6

- The conversion from kilometers to miles is (miles = kilometers * 0.621371). Write a for loop that converts 1 through 10 kilometers to miles and prints it to the console.

Example:

1 kilometer is 0.621371 miles

2 kilometers is 1.24274 miles

…

10 kilometers is 6.21371 miles

Do-While Loop

```
int i = 0;
do
{
   cout << i << endl;
   i++;
}while(i < 5);
```

Output:
0, 1, 2, 3, 4

- Do-While loop is the same as a while loop with the exception that the condition is evaluated at the end of the loop
- The significance of this is that a do-while loop guarantees at least one execution of the loop
- Notice the output of this loop is identical to that of a while

Do-While Loop Syntax Differences

```
int i = 0;
do
{
   cout << i << endl;
   i++;
}while(i < 5);
```

1

2

3

1. The use of the keyword do at the start of the loop
2. The while is at the end of the loop
3. There is a semicolon after the while

Do-While Functional Difference

```
int i = 5;
While(i < 5)
{
   cout << i << endl;
   i++;
}

Output: nothing
```

```
int i = 5;
do
{
   cout << i << endl;
   i++;
}while(i < 5);

Output: 5
```

- Here we note that since the while loop evaluates the condition at the start of the loop, while the do-while evaluates at the end of the loop the output varies slightly.

When to use a For Loop/While Loop

- For Loops excel at count control loops
  - Interest of x number of years
  - Decimal to Binary, Binary to Decimal
  - Calculate Powers
- While Loops excel at loops which have no counter and are waiting for a condition to change
  - Repeating Program
  - Timer

- For and While loops are interchangeable, use the loop that represents the code most clearly

When to use a Do-While

- When you want to execute the contents of the loop at least once
  - Repeating the program

- All Do-While loops can be implemented as for or while loops by setting up the variables in the condition to guarantee at least one execution
  - Much like the example of repeated triangle and rectangle calculation

Loops In Terms of Frequency of Use

1. For Loop
2. While Loop
3. Do-While

Breaks

- The keyword *break* terminates the execution of the nearest enclosing loop or switch statement in which it appears
- It has two uses
  - Exiting out of a loop prior to its natural end
  - Exiting out of a switch statement at the end of a case
    - Will be discussed in future

Impractical Break Example

```
int i = 0;
While(i < 5)
{
  cout << i << endl;
  i++;
  break;
}
cout << “done” <<endl;

Output:
0
done
```

- On encountering the break the loop is immediately exited
- Therefore only one iteration will ever be completed
- Notice it is an unnatural end to the loop since it does not exit on a condition evaluation

Practical Break Example

```
int i;
cin >> i;
While(i < 5)
{
  cout << i << endl;
  if(i == 2)
     break;
  i++;

}
```

- When breaks are used inside of loops they are always found within a if statement
- The if statement checks for a specific case where the code might need to exit
- In this case the loop will exit if a 2 is encountered, otherwise execute normally

- Often these are used for exiting in a loop if an error occurs
- Also can be used if the condition becomes too complicated

Continues

- The keyword *continue* immediately skips the rest of the current iteration of the loop to execute the next.

Impractical Continue

```
int i = 0;
While(i < 5)
{
  cout << i << endl;
  continue;
  i++;
}

Output:
0, 0, 0, 0, … 0
```

- On encountering the continue the current iteration stops, i.e. the i++ is not execute, and the next iteration begins
- Note: the evaluation will still occur

Impractical Continue with For

```
int i;
for(i = 0; i < 5; i++)
{
  cout << i << endl;
  continue;
  cout << “not executed”;
  }

Output:
0, 1, 2, 3, 4
```

- In the case of the for loop, the increment/decrement still occurs prior to the evaluation on encountering the continue

Practical Continue Example

```
int i = -1;
While(i < 9)
{
  if(i % 2 == 1)
  {
      i++;
      continue;
  }
  i++;
  cout << i << endl;
}
```

- When continues are used inside of loops they are always found within a if statement
- Here a continue is used to skip odd numbers

```
Output:
0, 2, 4, 6, 8
```

Ignore this slide

- The goto statement, used to jump to a label

```
int i = 0;
loop:
cout << i << endl;
i++;
if(i < 5)
      goto loop;
Output:
0, 1, 2, 3, 4
```

- Here it implements the functionality of a do-while loop
- Very old school style of coding

Forget this control structure exists. Seriously, if you use it I will deduct points. Its terrible.

The Switch Statement – The last control structure

```
int x;
cin >> x;
switch(x)
{
  case 1:
    cout << "x is 1" << endl;
    break;
  case 2:
    cout << "x is 2" << endl;
    break;
  case 3:
    cout << "x is 3" << endl;
    break;
default:
    cout << "x is not 1, 2, or 3" << endl;
}
```

- A switch statement is a special type of conditional statement in which variable is checked for equivalence against a series of constant literals

The Switch Statement – The last control structure

```
int x;
cin >> x;
switch(x)
{
  case 1:
    cout << "x is 1" << endl;
    break;
  case 2:
    cout << "x is 2" << endl;
    break;
  case 3:
    cout << "x is 3" << endl;
    break;
default:
    cout << "x is not 1, 2, or 3" << endl;
}
```

- x is compared to each case until a case matches. Then the case is executed and the break causes an exit to the switch statement.
- default is executed if no cases matched (similar to an else)
- default should always go as the last case, or not at all

Equivalent If-Else

```
switch(x)
{
  case 1:
    cout << "x is 1" << endl;
    break;
  case 2:
    cout << "x is 2" << endl;
    break;
  case 3:
    cout << "x is 3" << endl;
    break;
default:
        cout << "x is not 1, 2, or 3" << endl;
}
```

```
if(x == 1)
   cout << "x is 1" << endl;
else if(x == 2)
   cout << "x is 2" << endl;
else if(x == 3)
   cout << "x is 3" << endl;
else
   cout << "x is not 1, 2, or 3" << endl;
```

Properties of Switch

- Any switch can be implemented as a series of if statements
  - **Not** all if statements can be implemented as switch statements
  - Only if equivalence to a constant is being checked can a switch statement be used.
- Example of a if statement that cannot be implemented as a switch statement
  - Complex conditions
  - Ranges of numbers

```
if(1.5 < x && x < 10.3)
    ...
```

Property: Falling Cases

```
switch(x)
{
  case 1:
    cout << "x is 1" << endl;
  case 2:
    cout << "x is 2" << endl;
  case 3:
    cout << "x is 3" << endl;
  default:
        cout << "x is not 1, 2, or 3" << endl;
}
```

- In this example the breaks were removed from each case
- This will cause a given case to fall through to the next

Property: Falling Cases

```
switch(x)
{
  case 1:
    cout << "x is 1" << endl;
  case 2:
    cout << "x is 2" << endl;
  case 3:
    cout << "x is 3" << endl;
  default:
        cout << "x is not 1, 2, or 3" << endl;
}
```

Example: x is 2

Output:

x is 2

x is 3

x is not 1, 2, or 3

- Here case 2 is executed and executes "falls" through the other cases since no break is present
- Sometimes this is a bug, sometimes it is beneficial

Benefits of Falling Cases

```
char lttrGrade;
cout << "Enter the letter grade: ";
cin >> lttrGrade;
switch(lttrGrade)
{
  case 'A':
  case 'a':
    cout << "Falls between 90 and 100" << endl;
    break;
  case 'B':
  case 'b':
    cout << "Falls between 80 and 90" << endl;
    break;
... Excluded for Readbility ...
  case 'F':
  case 'f':
    cout << "Falls between 0 and 60" << endl;
    break;
  default:
    cout << "Not a letter grade" << endl;
}
```

- In the example, if lttrGrade is 'B' then *Falls between 80 and 90* is printed.
- This is because case 'B' falls into 'b' and is executed, but does not fall into case 'C' because of the break

Unguided Example 5.7

- Have the user enter in two numbers in which they wish to perform an arithmetic operation on.
- Prompt the user to select which arithmetic operation they wish to perform (-, +, *)
- Using a switch statement, perform the specified operation and print the result