PIC 10A 1C. Week 6a Notes. TA: Eric Kim

DeMorgan's Laws

DeMorgan's Laws are a convenient way of rewriting a logical condition to a different, yet logically equivalent, form. Sometimes it can simplify complicated expressions into a form easier to understand.
For the following, let S1, S2, …, SN be expressions taking on `true`/`false` values.

1. `!(S1 && S2 && … && SN)` ⇔ `(!S1) || (!S2) || … || (!SN)`
   Not all are true ⇔ At least one is false

Example: The following are logically equivalent:

```
int a = 2; int b = 6;
if (!( (a < 3) && (b > -5) )) {
    cout << 1;
}
```

```
int a = 2; int b = 6;
if (( !(a < 3) || !(b > -5) )) {
    cout << 1;
}
```

One can further simplify the right column since `!(a < 3)` is logically equivalent to `(a >= 3)`:

```
int a = 2; int b = 6;
if (( (a >= 3) || (b <= -5) )) {
    cout << 1;
}
```

2. `!(S1 || S2 || … || SN) <=> (!S1) && (!S2) && … && (!SN)`
   None are true <=> All are false.

Example: The following are logically equivalent:

```
int a = 2; int b = 6;
if (!( (a < 3) || (b > -5) )) {
    cout << 1;
}
```

```
int a = 2; int b = 6;
if (( !(a < 3) && !(b > -5) )) {
    cout << 1;
}
```

One can further simplify the right column as:

```
int a = 2; int b = 6;
if (( (a >= 3) && (b <= -5) )) {
    cout << 1;
}
```

while loops

A `while` loop is used to repeat a block of code multiple times. This programming construct is found in nearly every programming language, and greatly expands the expressiveness of our programs. Here's an example:

Code:	**Output:**

`int i = 0;`
`while (i < 4) {`
`  cout << "i is: " << i << endl;`
`  i += 1;`
`}`
`cout << "Here!";`
	`i is: 0`
	`i is: 1`
	`i is: 2`
	`i is: 3`
	`Here!`

One can also repeatedly interact with the user via a while loop:

```
// Keep asking user until user guesses correctly
int n = 1;
bool keep_looping = true;
while ( keep_looping ) {
  cout << "What is 40 + 2?" << endl;
  cin >> n;
  if (n == 42) {
    cout << "Correct!" << endl;
    keep_looping = false;
  } else {
    cout << "Nope! 40 + 2 is *not* " << n << "! Try again.\n";
  }
}
```

The syntax of a while loop is as follows:

```
while (EXPRESSION) {
    BODY
}
```

EXPRESSION is any expression that evaluates to a true or false value. The BODY of a while loop can be any number of statements. The while loop proceeds as follows:

1. Check value of EXPRESSION. If it is a false value, then exit the while loop.
2. Otherwise, run the BODY. Go back to (1).

break statement

One way to terminate (or "break" out of) a while loop is to use the break statement. For instance, the above while loop could be equivalently expressed as:

```
int n = 1;
while (true) {
  cout << "What is 40 + 2?" << endl;  cin >> n;
  if (n == 42) {
    cout << "Correct!" << endl;
    break;  // Terminates loop
  } else {
    cout << "Nope! 40 + 2 is *not* " << n << "! Try again.\n";
  }
}
```

Note: In the above, "`while (1)`" may make you feel a little nervous, since it appears that the while loop will never finish looping. Indeed, a common programming bug is to have while loops that endlessly loop (known as **infinite loops**). However, in this case it is ok, because we want to terminate the loop (`break`) only when the user types in 42.

Empty Statements

In C++, an empty statement is simply a lone semicolon:

```
if (3 == 3) {
    ;    // Do nothing
} else {
    ;;;; // Do lots of nothing :P
}
```

They are mostly harmless, and not often used, but you should at least be aware of them.

Comparison == vs Assignment =.

This error is common enough that we make a special note of it here.
To compare two values, we use the comparison operator ==, ie **two** equal-signs.
To assign a new value to a variable, use the assignment operator **=**, ie **one** equal-sign.

`int x = 4;`
`if (x == 2)`
`  cout << 1 << endl;`
`else`
`  cout << 2 << endl;`
`cout << "x is: " << x;`
	Output
	2
	x is: 4

`int x = 4;`
`if (x = 2)`
`  cout << 1 << endl;`
`else`
`  cout << 2 << endl;`
`cout << "x is: " << x;`
	1
	x is: 2

In the second example, because we used the assignment operator (=), we end up doing::
First, update x to be 2. Interestingly, the value of an assignment is the variable's new value. Thus, the if condition goes from "`if(x = 2)`" to: "`if(2)`". Since 2 is a true value, output 1.

!!!! NOTE !!!!: Midterm 1 does not cover past while-loops. Thus, you do not have to know do-while loops and for-loops for midterm 1.

do-while loops

Very similar to while-loops. A while-loop works as follows:

```
while (EXPRESSION) {
    BODY
}
```

1. Check the value of EXPRESSION. If it is false, then exit the loop.
2. Otherwise, run the BODY, then go back to (1).

On the other hand, a do-while loop works as follows:

```
do {
    BODY
} while (EXPRESSION);    // Note the semicolon!
```

1. Run the BODY.
2. Check the value of EXPRESSION. If it is false, then exit the loop. Otherwise, go back to (1).

Thus, the main difference is: a do-while loop will always run its body at least once, whereas a while loop may never run its body.

`int i = 2;`
`while (i < 0) {`
`  cout << "Hi!\n";  i = i + 1;`
`}`
`cout << "i is: " << i;`
	i is: 2

`int i = 2;`
`do {`
`  cout << "Hi!\n";  i = i + 1;`
`} while (i < 0);`
`cout << "i is: " << i;`
	Hi!
	i is: 3

for loops

Yet another way to write loops. A typical example is:

`for (int i = 0; i < 5; ++i) {`
`  cout << i;`
`}`
`cout << "\nDone!";`
	012345
	Done!

The syntax is:

```
for (INITIALIZATION; CONDITION; INCREASE) {
    BODY
}
```

When a for-loop is executed, the following happens:

1. The INITIALIZATION statements are executed, ie defining loop variables.
2. CONDITION is checked. If it is a false value, we exit the loop. Else, go to (3).
3. Run the BODY.
4. Execute the INCREASE statement(s), ie incrementing loop variables.
5. Go to (2).

Aside:Out of the three looping constructs (while, do-while, for), the most common are `for` loops, followed by `while` loops. I personally rarely encounter `do-while` loops.

Variable Scopes (while, do-while, for)

If a variable is defined inside of the body of a while/do-while/for loop (or within the for-loop initialization), then the variable only exists within the while/do-while/for loop body.

`int i = -1;`
`for (int i = 0; i < 3; ++i) {`
`  cout << i << endl;`
`}`
`cout << "after for-loop: " << i;`
	Output:
	0
	1
	2
	after for-loop: -1

`int i = -1;`
`for (i = 0; i < 3; ++i) {`
`  cout << i << endl;`
`}`
`cout << "after for-loop: " << i;`
	Output:
	0
	1
	2
	after for-loop: 3

`for (int i = 0; i < 3; ++i) {`
`  int x = 42; cout << x;`
`}`
`cout << x << i;`
	CompileError: x and i are not defined outside of the for-loop body!

In the first example, the for-loop creates a variable "int i = 0" that is *separate* from the variable "int i = -1". Thus, after the for-loop finishes, i is still -1.
In the second example, however, the for-loop initialization "i = 0" refers to the previously-defined variable "int i = -1".
In the third example, the variable x is defined only within the for-loop body, but not outside. Thus, we get an undefined variable error when we try to use x outside of the for-loop. In other words, the variable x only exists in the for-loop's **scope**.