PIC 10A 1C. Week 6b 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

<u>Example</u>: 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.

<u>Example</u>: 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 <u>block</u> 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:

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

Output:

```
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 (true)`" 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.

continue statement

In any looping construct, the continue keyword will cause the program to jump to the end of the loop body, skipping the rest of the body:

```
while (true) {
    cout << "Please type 42:" << endl;
    int n;  cin >> n;
    if (n != 42) {
        cout << "n is not 42, try again." << endl;
        continue; // redo loop
    }
    cout << "You did it!" << endl;
    break; // exit loop
}
```

The above code will keep asking the user to enter 42 until they do so.

Empty Statements

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

```
    if (1 == 1) {
        ;;;;;    // Do nothing
    }
```

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.

do-while loops

Very similar to while-loops. 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

Another popular way to write loops:

```
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.

Finally, you can declare multiple variables **of the same type** with the following syntax:

```
    for (int i=0, j=5; i < j; ++i) {
        cout << i << " " << j;
    }
```

Variable Scopes (while, do-while, for)

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

Range-For Loops

This is a convenient way to loop over all of the elements of some object, eg the characters of a string:

`string s = "cat";`
`for (char c : s) {`
`    cout << c << endl;`
`}`
	Output:
	`c`
	`a`
	`t`

The syntax is:

```
    for (TYPE VAR : OBJECT) {
        BODY
    }
```

The range-for loop is only appropriate for objects that "know" how to be iterated over. For instance, string objects know how to be iterated over: character by character. Soon, we'll learn about the vector class, which is an object that stores other things of the same type. Peeking ahead a bit, we can use the range-for loop to easily iterate over a vector of ints:

```
vector<int> myints = {3, 4, 5, 6};
for (int n : myints) {
    cout << n*n << " ";
}
```

Output:

```
9 16 25 36
```

If you want to modify the object you are iterating over, you can use references:

```
string s = "cat";
for (char &c : s) {
    if (c == 'a')
        c = 'o'
}
cout << s;
```

Output:

```
cot
```

Extra Remark: cin and loops

If you want to repeatedly ask the user for input until cin enters a failure state, here are two equivalent ways:

```
int n;
while (true) {
    cin >> n;
    cout << "You typed: " << n << endl;
    if (cin.good() == false) {
        break;
    }
}
```

```
int n;
while (cin >> n) {
    cout << "You typed: " << n << endl;
}
```

How is the right equivalent to the left? Well, it turns out that (cin >> n) returns a true value if a valid input is provided to cin. If the user types an invalid value, ie setting cin to a failure state, then (cin >> n) returns a false value.

Note: In Windows, to explicitly send cin into a failure state when it is waiting for input, you can type <CTRL>z<ENTER>.

Vectors

A vector is a resizable container that stores items of the same type. To use vectors in your program, include the <vector> library. For instance, to create a vector that stores integers:

```
vector<int> myints;  // []
myints.push_back(3); // [3]
myints.push_back(5); // [3, 5]
```

Creating Vectors

```
vector<int> v; // Creates empty vector
vector<int> v(5); // Creates vector with 5 ints
                  // default-initialized to 0
vector<int> v(3,2); // Create vector with 3 ints, all entries are 2
vector<int> v = {9,3,2}; // Creates vector with values: [9, 3, 2]
```

You can create vectors that store any type you like, including user-defined classes:

```
    vector<string> mysong = {"This", "is", "the", "story"};
    vector<double> nums = {1.2, 1.3, 42.9};
    // Assume Person class exists
    vector<Person> people = {Person("Troy"), Person("Abed")};
```

To access elements of a vector, use the [] operator. Remember to use 0-based indexing!

```
    vector<int> ds = {42, 9001, 1};
    cout << ds[0] << endl; // Outputs: 42
    cout << ds[2] << endl; // Outputs: 1
```

One can modify vector elements by using the [] operator as well:

```
    ds[1] = -9; // ds: [42, -9, 1]
    cout << ds[1]  << endl; // Outputs: -9
```

Vector Methods

`size_t vector::size()`
Returns the number of elements within the vector.

`void vector::push_back(val)`
Adds an element to the end of the vector. Grows the vector size by 1:

```
    vector<int> v2 = {5, 7, 11, 13};  // [5, 7, 11, 13]
    v2.push_back(17); // v2: [5, 7, 11, 13, 17];
```

`void vector::pop_back()`
Removes the element at the end of the vector. Shrinks the vector size by 1. If you call pop_back() on an empty vector, the code will crash!

```
    vector<int> v3 = {1, 3, 5}; // [1, 3, 5]
    v3.pop_back(); // v3: [1, 3]
    v3.pop_back(); // v3: [1]
    v3.pop_back(); // v3: []
    v3.pop_back(); // Crash!
```

`size_t vector::capacity()`
Returns the number of elements that this vector can store before it is forced to reallocate more memory. Will always be greater than or equal to its `size()`.

`void vector::reserve(size_t size)`
Asks the vector to (potentially) grow its internal container such that its capacity is at least `size`.

vector size() vs capacity()

At any given moment, a vector object has a certain amount of "slots" that are able to be filled by push_back() operations. For example, a newly-constructed empty vector object `vec` can still have 10 available slots. Here, vec.size() would be 0, but vec.capacity() would be 10.
Suppose we added 10 elements to vec. When we go to add a new element, vec will automatically ask for more slots to accommodate new entries. A typical strategy is to double the capacity each time, ie go from 10 -> 20 slots. Now, vec.size() would be 11, and vec.capacity() would be 20.
This is done purely for performance reasons - it turns out that increasing the capacity in this manner is significantly faster than incrementally increasing the capacity each time an element is added (the reasoning uses something called *amortized* analysis).
Finally, you can use `vector::reserve(size_t s)` to explicitly ask for the capacity to be at least s.

Nested Vectors

Since vectors can store any data type, you can also have vectors that store other vectors!

```
// Create two rows of three numbers, all initialized to 1:
//     1 1 1
//     1 1 1
vector< vector<int> > nums(2, vector<int>(3, 1));
nums[1][2] = 0;
// nums is now:
//     1 1 1
//     1 1 0
```

To understand how the assignment "nums[1][2] = 0" works, we can break it down:

```
vector<int> &row = nums[1]; // row: [1, 1, 1]
row[2] = 0; // row: [1, 1, 0]
// nums is now:
//     1 1 1
//     1 1 0
```

Take care that I made row a reference variable. The following does not modify nums:

```
// Create two rows of three numbers, all initialized to 1:
//     1 1 1
//     1 1 1
vector< vector<int> > nums(2, vector<int>(3, 1));
vector<int> row = nums[1]; // make a *copy* of nums[1], set to row
row[2] = 0; // modifies row, but not nums
// nums is still:
//     1 1 1
//     1 1 1
```

Note: A nested vector/array is also known as a 2D vector/array, or a matrix. The above nums example is a 2x3 matrix, since it has two rows and three columns. Think of nums as a vector of rows.
Warning: Be sure to include a space! "`vector<vector<int>> v`" is a compiler error! Instead, do:

```
vector < vector<int> > v;
```

Arrays

Arrays are like a primitive version of vectors. Like vectors, arrays are containers, yet there are some major differences:

Vectors	Arrays
- Are objects (v.size(), v.push_back(), etc.). - Can grow/shrink dynamically	- Not objects. Can't use dot notation! - Array size is fixed at creation (compile) time.

Array Creation

Since array sizes are fixed, you must specify the size of an area at compile time. In particular, if you declare the size of an array in a variable, that variable must be declared const:

```
const size_t maxCapacity = 10;
int myArray1[maxCapacity]; // Create array of size 10
int myArray2[10]; // Create array of size 10
int size_t maxCap2 = 20;
int myArray3[maxCap2]; // CompileError! maxCap2 not const.
int thisArray[] = {1,2,3,4}; // Create array: [1, 2, 3, 4]
int a[6] = {1, 0, 2}; // initialize only first three values
```

One can access and modify elements of an array via the [] operator:

```
int a1[] = {1, 2, 3};
cout << a1[0] << endl; // Outputs: 1
a1[2] = -42;
cout << a1[2] << endl; // Outputs: -42
```

Finally, like vectors, you can put any data type inside of an array: ints, doubles, strings, vectors, even other arrays!

```
string wds[] = {"hi", "there"}; // array of strings
int[2][3] nums; // two rows, each row having three values
nums[0][2] = 42; // assignment works similarly to vectors
```