PIC 10A 1C. Week 8a Notes. TA: Eric Kim

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 = {1, 2}; // Creates vector with values: [1, 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!
```

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.

Warning: Be sure to include a space! "`vector<vector<int>> v`" is a compiler error! Instead, do:

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

Copy Assignment

In C++, whenever you assign a variable without the reference operator, you actually make a copy of the right hand side. Thus, when I do "vector<int> row = nums[1];", the following happens:

(1) Make a copy of nums[1], ie a new vector with values [1,1,1]

(2) Assign `row` to the new copy.

Aside: This is a simplified view of what actually happens. In C++ there is the copy constructor and the assignment operator, both of which perform copying depending on the situation. We haven't covered this yet (not sure if we will), so don't worry about this distinction.

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
```

According to Professor Lindstrom, we won't go into too much detail with arrays. If you like, you can view them as a historical predecessor to vectors.

Random Numbers

To generate pseudo-random numbers, you can use the rand() function.

`int std::rand()`

Outputs a pseudo-random integer between 0 and RAND_MAX (inclusive).

```
for (int i = 0; i < 4; ++i ) {
    cout << rand() << endl;
}
```

Output:

```
41
18467
6334
26500
```

Rather than output truly random numbers (difficult to do on deterministic machines such as digital computers), the rand() function instead outputs numbers from a **pseudorandom number generator** (PRNG). A PRNG is designed to output values that, for all intents and purposes, appear to be randomly generated. However, in actuality a PRNG is actually *deterministic*, and will always output the same numbers for a given initial **seed**.
A PRNG is initialized by giving it a **seed** number. The PRNG uses this initial seed value as a "starting point" for generating its pseudorandom numbers. One can consistently output the same pseudorandom values by using the same seed:

`srand(42);`
`for (int i = 0; i < 4; ++i ) {`
`  cout << rand() << endl;`
`}`
`srand(42);`
`for (int i = 0; i < 4; ++i ) {`
`  cout << rand() << endl;`
`}`
	Output:
	175
	400
	17869
	30056
	175
	400
	17869
	30056

`void std::srand(unsigned int seed)`
Sets the seed of the pseudorandom number generator.

One popular way of seeding the PRNG is to use the current time:

```
srand(time(nullptr)); // seed PRNG with current time
```

`time_t std::time(time_t* timer)`
Returns the current calendar time, updating the timer argument if it's not the null pointer. In particular, this returns the number of seconds since 00:00, Jan. 1st, 1970 UTC (ie the unix timestamp). This is a function in the <ctime> library.

The nullptr, or "null pointer", is a pointer that points to nothing. Here, it is used to signal to time() that we are not passing in a time_t timer. We'll learn more about pointers soon.

Finally, RAND_MAX is a constant defined in <cstdlib>, and is the largest number that rand() can output. We can ask C++ to output RAND_MAX for us:

```
cout << RAND_MAX << endl; // Outputs 32767 on my laptop
```

Note: rand is defined in the <cstdlib> library.

Handy Tip: To output a random number between [a,b] (inclusive):

```
int a = 1; int b = 10;
cout << (rand() % (b-a+1)) + a << endl;
```