CSI 201
learning vectors and lists, includes and namespace

1. Goals for today: learning vectors with some insight into includes and namespaces!

2. If you want to skip straight to lists, move to the next page!

3. So how do we make lists in C++? First, let's discuss some of the constructs we've mostly been avoiding. The first is the `#include`! The include pieces of our code insert code into our source code during one of the first steps in the compiling process. If we wanted, we could actually go look up precisely what exists in the "iostream" file. However, this is not something software developers generally need to do with system includes like this. The main point is though, that they include some code for us. During the linking step of the compiler, the functionality that those libraries provide is connected to our program in one way or another. Static linking includes all the code very directly into our executable. Our code might instead, dynamically link to a library. In windows, we have DLL (dynamic link library) files. These are basically just suites of code that our program can connect to when it runs to accomplish some tasks. If you're interested in a bit more about why this is a good idea and is used in a lot of operating system, then please please take some operating systems! We talk about this here, because vectors, (like strings and iostream and a few other things we'll learn soon) need a new include. To use them, we must:
   ```
   #include <vector>
   ```

4. In addition, we also use `using namespace std;` at the end of our includes. We use this to connect to the `std` (or standard) versions of the components that we're including. It is possible to include two libraries that both use `cout` but implement them slightly differently. In such an event it is important to distinguish between which type our code should understand and utilize. In industry, it is thus usually standard not to use `using namespace` in our code and instead, to refer to objects by their namespace and name, like: `std::cout`. However, for learning to program this can be a bit cumbersome. So we will keep on using the std namespace for the rest of the semester. DLLs, namespaces, and includes are super cool and worthy of our note and time. However, we're going to spend more time to understand basic code pieces and leave some of this concepts to higher level courses or to be picked up separately.

5. Remember how we repeat tasks many times:

```
for(int i = 0; i < 10; i++) { //finally the reason!
    //this task happens 10 times
    //but with different values of i!
}
```

6. We'll be using that for loop from above with our new list construct. Here each `i` will be the index of an element of the list we want. So if the list contains [10, 33, 42] then, the element at index 0 is 10. The element at index 1 is the second element and it is the number 33. The last element is at index 2 (which is one less than the size of the list) and it has the value 42. We take note that the way we have been writing for loops facilitates this indexing. That for loop above produces an i that changes from 0 to 9. That would be perfect for the indices of a list with 10 elements.

7. Don't forget, to use vectors, we must `#include<vector>`. Okay, let's get back to talking about lists! We want to make some. In C++, the basic type of list is called an array. However, it is not very dynamic. So, we're going to talk about the slightly easier topic: vectors! Vectors are a type of list that must all consist of the same type of object. So we can create a **vector of integers** or a **vector of strings** but we cannot create a vector of mixed integers and strings (well at least not without some more framework from deeper in this course). But, after we make our list (let's call the list names), we can perform 3 basic operations. Let's examine two ways to make a list and these operations:

   (a) `vector<string> names;` – Makes a vector of strings. There are no names in this list yet. The size of the list is 0. This is kind of like having an empty line at the grocery store. if there was a sign, it would say "there are 0 people in this line."

   (b) `vector<string> names(unsigned);` – This is another way to make a vector of strings. However, this time, we are telling the vector how many elements are in the list very directly. Sample code might be: `vector<string> names(10);` This would create a list of 10 strings. For strings, we know they are all empty strings, but if we made a list of other objects, (like integers), we might not necessarily know what value is in each element of the list. So it is a good idea to write a loop to initialize these variables directly and immediately. Also, since the variables already exist, we can immediately access these using the `at` functionality from below. Also note: If we use push_back (from below) on this list, it will add MORE elements to the end of the list. A common mistake is to make room for variables but then add to the end.

   (c) `names.size()` – get the size of the list as an unsigned

   (d) `names.at(int)` – use an integer to get the value of the element at that specific index. For example, `names.at(0)` is the exact same as using the variable that would exist at the 0 index of the list. We can use `cout`, assignment or any other arithmetic with it that we would like.

(e) `names.push_back(string)` – since names is a list of strings, we can add (push) strings to the end (back) of the list. This is where the name push_back() comes from! To do so, we must actually add a string. So for example, if we wanted to add "Rams" to the end of our list, we write: `names.push_back("Rams");`

8. So how do we make a variable to hold a list of things. Let's start with a list of strings. Explanation of the different things we can do with vectors will be explained in the code again below!

```
#include<iostream>
#include<string>
#include<vector>
using namespace std;
int main() {
  //declare a vector of strings to hold our list of names
  vector<string> names; //this is our variable - names

  for(unsigned i = 0; i < 300; i++) { //runs 300 times
     string user_input_name; //declare this variable
     cin >> user_input_name; //get a name from user

     //the next piece puts the name we got from the user
     //and adds it to the end of the names list
     names.push_back(user_input_name);
  }
  //when we get to this line in the code, we have a list
  //that contains 300 names that were typed by the user
  //so let's output the last 10 of them!
  //to do that we need to be able to get the size of the list
  //or at least, this allows us to do things generally, which
  //is better programming practice in the end. We do this by
  //taking the variable and adding .size() to the end. For Ex:
  unsigned size = names.size();

  //We must also access elements via index. To get access to
  //any of the elements we use .at(index) at the end of the
  //variable name. So, the following example takes the size
  //variable declared and initialized from the vector above,
  //to output the last 10 elements of the list using .at(int)
  for(unsigned i = size-10; i < size; ++i) {
     cout << names.at(i) << endl;
  }
}
```

9. Can we see how the code outputs only the last 10? Can we change it to do the first 10 only?

10. Can we change the code to output every 10th element?

11. Let's write some code to read in a list of grades (as integers). However, let's allow the user to determine when we are "done" rather than pre-determining the exact number of elements in the list. This will move us from using a for loop for input, to a while loop. Allow the user to keep inputting numbers until a negative number is input. Then, when complete, output the whole list!

12. This new thing we're making, like `vector<int> grades;` is actually its own new type. `vector<int>` is a type in its own right. It is the "vector of integers" type. Because it is a type, it can be used in vectors. We can make a vector of any type and thus we can make a "vector of vectors of integers". So we can create rows and columns of things in certain situations. We won't use these in this class, but what fun!

13. Other notes! vectors are part of STL. STL stands for standard template library. STL has a bunch of other containers for doing cool things for a variety of situations. Some containers include: stacks, queues, maps (dictionaries), lists, deques, sets and more. Generally, a good understanding of vectors can get us to a place where we can use any of the others. We should really check out dictionaries if we have more time. They allow us to make key-value pairs of any types we want. So whether we want to represent key-bindings in a game, map users to their network cards or map users to hashed passwords, dictionaries will work. STL has other things too (algorithms and the like), but we won't focus on those. Template is a really useful too that is also a little too advanced for an introductory class in programming. But it is a neat way to write single pieces of code that apply to many types. This is how we can create a vector of any type (because of templates!)