Namespaces

- We have seen expressions such as `std::cout`, that are unique to C++
- `std` refers to a `namespace`
- C++ programmers use code written by others, and names can easily conflict
- Namespaces allow distinguishing names declared in different files
- Still, it would be nice to not have to explicitly mention them all the time!
The **using** Directive

- Form: 
  `using namespace namespace-name;`
- Allows use of names declared within a namespace, without specifying the namespace each time
- Compiler will look in the namespace for names that are not declared elsewhere
- Example: **using namespace std;** allows use of `cout` instead of `std::cout`

---

**using** Declarations

- Form: 
  `using namespace-name::var-name;`
- Specifies that any instance of `var-name` (without a namespace) refers to the object `var-name` declared within `namespace-name`
- The **using directive** allows the compiler to look for any name in a namespace, but a **using declaration** forces use of a namespace for a specific name
Global Names

• Usually, variables are declared within a function (these are *local variables*) or within a namespace
• *Global variables* are declared outside of any function or namespace
• If a global variable conflicts with a namespace variable or local variable, we can prefix the global variable with a `::` (by itself) to avoid ambiguity

Namespace Aliases

• A namespace can have a very long name, for example to identify the company who created it
• A *namespace alias* allows a shorter name to be substituted for convenience
• Form:
  ```
  namespace alias = original;
  ```
• Then *original*: `x` can be replaced with `alias`: `x` throughout
Input from std::cin

- `char c; c = cin.get();` reads a character from `cin` and assigns it to `c`
- `string s; getline(cin, s);` reads a line from `cin` and stores it in `s`
- `cin.clear();` resets `cin` to working state after `>>` fails due to bad input
- `cin == true` is true if `cin` is in working state, false otherwise
- `cin >> x` reads from `cin`, stores in `x`

Output to std::cout

- `cout.put(c);` writes `char c` to `cout`
- `cout << x` writes `x` to `cout`
- `cout.flush();` forces any buffered output to be written immediately
- `cout.width(n);` sets field width to `n`
- `cout.precision(n);` sets number of decimal places printed to `n` (default 6)
- `cout.fill(c);` sets the fill character to `c` (default is space)
**std::string**

- In C, strings are arrays of characters
- The C standard library provides functions for working with strings, but they are still cumbersome
- The type **std::string** provides higher-level functionality
- **strings** can be initialized at declaration (like integer variables), and assigned, unlike in C

---

**Operations on strings**

- Can access individual characters of a string using the [] operator, which also accesses elements of an array (discussed next time)
- Can use + to concatenate **string** objects, as well as string literals
- Can use >> to read from an input stream into a **string** object, but only one word at a time
std::vector<>

- This is an incomplete type, known as a *class template*
- The type is completed by specifying the type of the elements of the vector, between the <>
- Generic operations are implemented once, and apply to vectors of any type
- Vectors can be treated as a list, or as an array (for random access), depending on the application

---

Example: Sorting

- Reads numbers from standard input
- Stores them in a `vector` object
- Uses `sort` function to sort them in increasing order
- Vector is sorted "in-place", that is, overwritten with elements in sorted order
- Iterates through the vector, using random access to display each one
Iterators

• An iterator is a type whose values locate objects as if they were C array elements
• Types of iterators:
  – Trivial: gives location and nothing more
  – Forward: can use ++ to access next location
  – Bi-directional: Forward, along with --
  – Random-access: Bi-directional, with +, – to skip over locations, [] to access any location
• sort uses random-access iterators from begin() and end() member functions

Next Time

All about functions!
• Types of functions (this bullet point has a double meaning)
• Overloading operators
• inline functions
• Passing arguments