Friends
- Another variation of access control
- A class may declare a non-member function to be its friend:
  - Keyword friend begins the declaration
  - Otherwise the declaration is a function prototype
  - Friend declaration is within the class definition
  - Function declared a friend is implicitly a non-member
- A whole class may be declared as a friend
  - Syntax is friend class_name;
  - May require forward declaration of class class_name for mutual friends
- Friends have all the access privileges of member functions
- Breaks down encapsulation
  - But preferable to making data completely public
  - Specification is in the class definition, so is at least obvious
  - Specific to a function or class, so should be no surprises
- Useful for closely coupled classes when nesting is not appropriate
  - Also can be used to make a nested class a friend
- Should be used sparingly and only as a reflection of design

Standard Template Library (STL)
- Standard C++ library of generic container classes
  - vector (like an array, but grows automatically)
  - set (no duplicates)
  - list
  - queue
  - deque
  - stack
- Some common operations
  - size(), empty(), swap(), ==, !, <
- Some operations specific to container
  - [] (all but list)
  - front(), back()
  - push_back(), pop_back(), (all)
  - push_front(), pop_front() (list and deque)
  - Dealing with the event of the exception (catching the exception)
- Containers work for any type, but are homogeneous for the type
- Containers use iterators to traverse the elements
  - Iterators are generic nested classes
  - Initialized with begin(), end(), rbegin(), rend()
  - Operator ++ increments to point to “next” element
  - Operator * dereferences to get the element value
- Example
  ```c
  #include <vector>
  
  void print(const vector<int> & v) {
    cout << "[ ";
  ```
for (vector<int>::const_iterator iter = v.begin();
    iter != v.end();
    ++iter)
    cout << *iter << " ";
cout << "]" << endl;
}

int main() {
    vector<int> v;
    for (int i = 0; i < 10; ++i)
        v.push_back(i*i*i);
    cout << "v.size()= " << v.size() << endl;
    print(v);
    cout << "[ ";
    for (int i = v.size() - 1; i >= 0; --i)
        cout << v[i] << " ";
    cout << "]" << endl;
    v[5] = 99;
    print(v);
    return 0;
}

- Other iterators – reverse_iterator, const_iterator

Namespaces
- Conserves the global name space
- Kind of like package in Java, but no access implied
- A namespace defines a scope
  - Then name and scope operator (::) is required, or
  - using statement has been given
- For example, using namespace std; avoids having to qualify cout
- A namespace is created using keyword namespace and a name for a block

Explicit Constructors
- Recall that constructors (with one argument) are type converters
  - They indicate how to build the class object from a value
  - Implicitly used for conversions
- Sometimes we don’t want this implicit conversion behavior
  - E.g., an Array with an integer size argument to a constructor
  - Does not mean we want an integer converted to an Array
  - But the compiler would do this if an Array was needed and an integer was what was available
- Avoid implicit conversion with explicit keyword
  - Qualifies constructor
  - Means constructor will only be used for declared values or new
  - Not used for implicit conversion
Variable Length Argument Lists
- This is mostly a legacy issue from C
- Classic example: printf, scanf
- Variable number of arguments
  - No type checking of arguments (or even count of arguments) possible
  - Trusts that function is invoked properly
  - E.g., in printf, the format argument says how many and what to expect
  - But compiler can’t tell what arguments to expect
- Syntax of declaring a variable argument function elides the arguments with …
- Coding the handling of variable arguments requires `<stdarg>` header (or `varargs.h` in older compilers)
- Several macros to help process arguments:
  - `va_list` Type used for object to access arguments
  - `va_start` Set things up
  - `va_arg` Get another argument to be accessed by `va_list`
  - `va_end` Clean things up
- Example:
  ```
  string mycat(int n, ...) {
    string result = "";
    va_list alist;
    va_start(alist, n);
    for (int i = 0; i < n; ++i)
      result += va_arg(alist, const char *);
    va_end(alist);
    return result;
  }
  ```
  ```
  int main() {
    cout << mycat(3, "hello", ",", "world") << endl;
    cout << mycat(5, "not", "with", "standing", "your", "objection") << endl;
    return 0;
  }
  ```
- Not very safe
- Can usually get type checking and same behavior by defaulting arguments or overloading various forms
- Can be useful for printf-like functions that are assumed to be used properly
- In example above, no type checking, so can’t use string in `va_arg` unless we know a string had been given