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

_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
  o No type checking of arguments (or even count of arguments) possible
  o Trusts that function is invoked properly
  o E.g., in printf, the format argument says how many and what to expect
  o 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:
  o `va_list` Type used for object to access arguments
  o `va_start` Set things up
  o `va_arg` Get another argument to be accessed by `va_list`
  o `va_end` Clean things up
- Example:
  ```c
  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;
  }
  ```
  ```c
  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

**Dynamic cast**
- Typical casting (static cast) is done at compile time. The compiler permits downcasting where a pointer can be cast to a sub-class pointer type (but if the object is not that actual type, bad things will happen at runtime). Upcasting is more common, where a pointer to a subtype may be treated as a pointer to a supertype (and no explicit cast is required to do this).
- Dynamic casting uses runtime type identification (RTTI) to permit casting to sibling types. This only works with polymorphic types (containing virtual functions)
- Syntax is: `A* ptr_to_A = dynamic_cast<A*>(some_ptr);`
- The cast will fail (produce a NULL result) if `some_ptr` is not a pointer to an A object (i.e., either it is an A object or uniquely inherits from one)