Overloading operators

● Why overload operators? (==, =, <, >>, +=, …)
  ➤ notational convenience
  ➤ match user expectations
  ➤ because we can (except :: and . And .*)
● C++ heuristic: avoid behind-the-scenes conversion
  ➤ Cpstring v. standard string

```cpp
string filename=“sesame.txt”; ifstream input;
input.open(filename);    // CPstring
input.open(filename.c_str());    // standard string
```

➤ vectors
  ```cpp
  Vector<int> a(20);
a[0] = 1;
a = 2;     // avoid with explicit keyword
  ```
● [www.cs.duke.edu/csed/cplus](http://www.cs.duke.edu/csed/cplus) and Stroustrup Chapter 11
motivation for Membersort<...>

class Alphasort : public Compare
{
    public:
    virtual int operator()(const DirEntry & lhs,
                           const DirEntry & rhs)
    {
        if (lhs.name() == rhs.name()) return 0;
        else if (lhs.name() < rhs.name()) return -1;
        else return 1;
    }
}
class Sizesort : public Compare
{
    public:
    virtual int operator()(const DirEntry & lhs,
                           const DirEntry & rhs)
    {
        if (lhs.size() == rhs.size()) return 0;
        else if (lhs.size() < rhs.size()) return -1;
        else return 1;
    }
}

● what about sorting by time? DirEntry::time()
Factor out common code, why use a template?

class Alphasort : public Compare
{
    public:
        virtual int operator()(const DirEntry & lhs, const DirEntry & rhs)
        {
            if (lhs.name() == rhs.name()) return 0;
            else if (lhs.name() < rhs.name()) return -1;
            else return 1;
        }
}

template <class Kind>
class MemberSort : public Compare
{
    public:
        typedef Kind (DirEntry::* MemPtr)() const;
        MemberSort(MemPtr ptr) : myPtr(ptr){}
        virtual int operator()(const DirEntry & lhs, const DirEntry & rhs)
        {
            if (((lhs.*myPtr)() == (rhs.*myPtr)()) return 0;
            else if (lhs.name() == rhs.name()) return 0;
            else if (lhs.name() < rhs.name()) return -1;
            else return 1;
        }
};
Alternatives to `makeIterator()`

- **return a vector**
  
  ```
  Scandir scanner;  Vector<DirEntry> vde; .... scanner.get(vde);
  ```

- **return a C-style array**
  
  ```
  Scandir scanner;  DirEntry * varray = scanner.get();
  ```

- **other options?**

- **advantages/disadvantages compared to iterator?**
- **problems with iterator, why `IterProxy`?**
- **where is the concrete iterator implemented, why?**
- **is friendship required?**
IterProxy

- Proxy aka surrogate (GOF 207)
  - need: more versatile or sophisticated reference to an object than a simple pointer
  - smart pointer, virtual proxy/web proxy

```c
PointerProxy<Iterator<DirEntry> > iter(scanner.makeIterator());
for( iter->first(); ! iter.isDone(); iter->next() )...
```

- note: iter is allocated on the stack, not on the heap
  - cleans itself up with destructor
  - acts like a pointer, overload operator->
  - potential problem with assignment of proxies
Assignment problems

IterProxy<DirEntry> iter(scanner.makeIterator());
IterProxy<DirEntry> copy = iter;
....
// both go out of scope

- options available to stop double deletion
  - disable assignment, copy --- how?
  - transfer ownership on assign or copy, but what to leave behind?

- other proxy uses include lazy/on-demand creation, copy on write, e.g., reference counted strings