Multiple inheritance

- Can derive a class from more than one base class
- e.g., `class Appliance;`

```
class Radio : virtual public Appliance;
class AlarmClock : virtual public Appliance;
class ClockRadio : public Radio, public AlarmClock;
```

  - A `ClockRadio` is both a `Radio` and an `AlarmClock` – so it is also an `Appliance`
  - Note `virtual` – just 1 `Appliance` subobject, not 2
  - See `ClockRadio` example: `…/demo08/multi-inherit`
- But note: hierarchy is messed up – best to avoid

Aside: how to safely read data

- Execute `…/hw4/array` – enter `"junk"` (after 1st iteration)
  - Use `ctrl-C` to stop the infinite loop, because that program is not "crash-proofed" at all
- See a better way in `…/demo10/goodflush/`
  - If bad data, clear error bits in cin – `cin.clear()`
  - Then must remove the bad data from the input stream
    - e.g., read it into a C string like the example programs
- Note: Nagler technique (p. 443-4) doesn't work
  - Becoming a theme? Try it in `…/demo10/badflush`

Other C++ input/output notes

- `std::ios_base` – atop the `iostream` hierarchy
  - Many public constants and functions
  - e.g., `cout.width(5); cout.setf(ios_base::right); …`
- `#include <sstream>` – for string streams
  - `ostringstream oss;` // now use `oss` just like `cout`
    - Get the string when done – e.g., `cout << oss.str();`
- Can do character I/O just as easily as in C
  - Use `cin.get()` and `cout.put(char)`
- Line input is easy too – good for crash-proofing
  - Use `cin.getline(`*`C-string, size [, delim_char]`*`)`

More C++ I/O notes

- Manipulators – functions with special signatures that are invoked by << and >>
  - Lots of built-in manipulators

```
cout << right << setw(5) << 'a' << endl;
```

  - Easy to write your own too – chapter 16 shows how
- File I/O – use `ifstream` and `ofstream` objects
  - Once opened, treat like `cin` and `cout`, respectively
  - Simplest way is to open on construction
    - `ofstream out("myfile"); out << "my data\n";`
  - Easy to learn other file techniques from chapter 17

`std::string`

- Object-oriented way to deal with character strings
- Actually defined type for `basic_string<char>`
  - Other: `typedef basic_string<wchar_t> wstring;`
- Must `#include <string>` to use these types
- Most of the features of Nagler's `class String`
  - And more: overloaded ops `=`, `+`, `+=`, `[]`, all relational ops, plus `insert()`, `substr()`, `getline()`, …
  - No conversion op – use `c_str()` function to get `char*`
- See `stringDemo()` function in `~cs60/demo10/librarytools.cpp`

Standard template library (STL)

- A framework of generic containers and algorithms
  - *STL containers are class templates* – for storing and accessing parameterized data types
  - *STL algorithms are function templates* – mostly involving contents of STL containers
- Iterators are the framework's linchpins
  - Essentially pointers to container elements
    - In fact, pointers into arrays can usually qualify
  - Each container type has a set of possible iterators
  - The algorithms access container elements using these iterators – so their use is standardized across containers

STL sequence containers

- `vector<typename>` – basically a smart array
  - Can even access random elements with `[]`
  - Unlike arrays, `vectors` grow dynamically as required, and have methods like `size()`, `empty()`, `clear()`, `insert()`, …
  - See `vectorDemo` function in `librarytools.cpp`
- `list<typename>` – a double-linked list
  - Quick insertion and removal of elements
  - No random access – but has bi-directional iterators providing access relative to existing elements
- `deque<typename>` – a vector/list combination

Adaptive sequence containers

- Underlying data structure is other sequence
  - But access is restricted in some defined way
- `stack<typename>` – LIFO access
  - Basic operations are `push()`, `pop()`, and `top()`
- `queue<typename>` – FIFO access
  - Operations are `push()`, `pop()`, and `front()`
- `priority_queue<typename>`
  - `push()`, `pop()`, and `top()` (like stack, not queue)
    - But pop() and top() access highest priority element

Associative containers

- Designed for accessing data by search keys
  - Main feature – quick `insert()`and `find()` operations
  - Also feature a natural ordering of the data elements
- Sets – the data *are* the keys
  - `set<typename, functor>` – no duplicates
    - The functor is used to order the elements
  - For duplicates: `multiset<typename, functor>`
- Maps – elements are key/data pairs
  - `map<keyT, dataT, functor>`, or allow duplicates with `multimap< keyT, dataT, functor>`

STL algorithms

- Function templates – mostly work with iterators
  - Idea – alternative to algorithms built into containers
    - Facilitates consistent handling of the various containers
- Usual: `alg(iterBegin, iterEnd,` *other args*`)`
  - e.g., `fill(vector.begin(), vector.end(), 0);`
  - e.g., `random_shuffle(v.begin(), v.end());`
  - More examples in `…/demo10/librarytools.cpp`
- One last thing – complete STL documentation is available online at http://www.sgi.com/tech/stl/

Unix shells

- Unix systems come with a variety of shells
  - Most common: Bourne (`sh`), C (`csh`) and Korn (`ksh`)
  - Newer: Bash (`bash`) and TC (`tcsh`)
- Primary purpose – interpret user commands
  - So a.k.a. command interpreters
  - Essentially *interfaces* to Unix kernel
    - The kernel is the actual operating system program
- Also programming languages in their own rights
  - Shell programs – series of shell commands, but also variables, conditional branching, loops, functions, …
  - A.k.a. scripts – are interpreted languages

Bourne shell programs

- Are text files with `sh` commands – e.g., `myScript`
  - To execute, can do `sh myScript`
    - The program runs in a new shell – called a child shell
  - Or `chmod u+x myScript` – then just `myScript`
    - But might not work if `sh` is not default shell
- `#` – usually identifies a comment
  - Special case if line 1 – `#!/bin/sh` – identifies shell
    - Means use `sh` as child shell for this script – works in all shells
- Can access command line arguments: `$1` to `$#`
  - e.g., `cp $1 $2 #` copies first to second (if files)
  - e.g., `echo $# #` prints number of arguments

sh variables and assignment

- `name="Jack Sprat"` # note no spaces
- `echo "The name is $name"` # need '$'
- `workdir=`pwd`` # use `…` to assign result of …
  - Similarly, `echo "date and time is `date`"`
- Can read from standard input and calculate too
  - `echo "enter value"`
  - `read val`
  - `doubleval=`expr $val + $val``
    - Or just: `echo "doubled: `expr $val + $val`"`

sh control structures

- An `if-then-elif-else-fi` statement
  - Expression is a test: `test $# -gt 0`
  - Or simpler: `[ $# -gt 0 ]` # spaces mandatory
  - Can test files too: `-d`, `-f`, `-e`, `-r`, `-w`, `-x`, …
- A `while` statement – same expressions
- A `for` statement – `for variable in list`
  - List is command line arguments if not specified
- See `~cs60/demo10` for examples
  (`sh` examples have "demo" in filenames)

C shell programs

- `csh` – can look a bit more like C programs
  - e.g., use `$argv[1]` instead of `$1`
  - `if-then-else-endif` – i.e., no `fi` or `elif`
    - Expressions more natural too: `()`, `==`, `>`, `||`, `&&`, …
  - `while` and `switch` structures also more like C
- Some things weirder than Bourne shell though
  - Need `set` for assignment, or `@` if numeric
  - Reading input is awkward: `set x = `head -1``
- But can use arrays, and system calls easier
  - See `*.csh` examples in `…/demo10`
- Learn about shell syntax with man – csh, sh, …

Perl – a *command language*

- Not a shell – but interprets *and compiles* scripts
  - Compiles at start of execution – to run loops much faster
- Written by a linguist, not a CS person – Larry Wall
  - Practical Extraction Report Language – more versatile than shell scripts, but less complicated than C programs
- `print "hello world.\n";` # note C-like syntax
- `$name = "John Smith";` # need `'$'` even for first use
- Has arrays, and *associative arrays* (lookup tables)
- Also string operators; C-like `if`, `while`, `for`; file I/O; functions; and access to library functions
- See `…/demo10/loan.pl` (many more at `www.perl.com`)