Reminder from last week …

Defining class templates

- Idea: “generalize” data that can be managed by a class

```
template<typename T>
class Pair {
public:
    Pair();
    Pair(T firstVal, T secondVal);
    void setFirst(T newVal);
    void setSecond(T newVal);
    T getFirst() const;
    T getSecond() const;
private:
    T first; T second;
};
```

Class template member functions

- All methods need template prefix – e.g., constructor:

```
template<class T>
Pair<T>::Pair(T val1, T val2)
    : first(val1), second(val2) { }
```

- Similarly setter and getter functions:

```
template<class T>
void Pair<T>::setFirst(T newVal)
{    first = newVal;   }
template<class T>
T Pair<T>::getFirst() const { return first; }
```

- See `~mikec/cs32/demos/templates/complex` example

Note: each function definition is itself a template

More class template notes

- Mostly design just like any class
  - Can have friends – usually do
  - Can be a base class or a derived class
- Careful though: `MyTemplate<T1>` ≠ `MyTemplate<T2>`
  - That is, there is no inheritance or any other kind of formal relationship between the two classes
    - e.g., cannot cast an object of one to an object of the other
  - Why?
    - Compiler defines completely different classes!

Class templates in OO design

- An alternative to using an inheritance hierarchy
  - More flexible, as template classes stand alone
  - More efficient than using virtual functions
- Both are ways to have objects with independent behaviors, but all sharing a common interface
- The STL is mostly template classes and functions
  - Ditto the Java Collections Framework by the way
- Even a `string` is actually a specialization of a template, defined as follows in `namespace std`:
  - `typedef basic_string<char> string;`
  - Also: `typedef basic_string<wchar_t> wstring;`

Starting Savitch Chapter 18

std::string

- *Encapsulates* a sequence of characters
  - i.e., much more object-oriented than `(char *)`
- Both a size and a capacity (for efficiency)
  - Both are mutable, and so are the characters
- Member operator functions `=`, `+=`, `[]`
- Others include `substr`, `insert`, `compare`, `clear`, …
- Nonmember: `op<<`, `op>>`, `getline`, `op+`, `op==`, …
- See http://www.cplusplus.com/reference/string/ and `librarytools.cpp::stringDemo()` in `~mikec/cs32/demos/templates/`

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 usually qualify for the functions
  - 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
  - Overloaded `[]` makes it seem like an array once created
  - But unlike arrays, `vectors` grow dynamically as required, and have methods like `size()`, `empty()`, `clear()`, `insert()`, …
- `list<typename>` – a double-linked list
  - Best feature: quick insertion and removal of elements
  - But no random access – must settle for using bi-directional iterators that provide access relative to existing elements
- `deque<typename>` – a vector/list combination
- See three related demo functions in `librarytools.cpp`

Adaptive sequence containers

- Underlying data structure is another sequence
  - With access 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()` (more like a stack than a 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
- Sets – the data *are* the keys
  - `set<typename, functor>` – no duplicates allowed
    - The "functor" (function object) is used to order the elements
  - To have 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());`
  - Demos: `~mikec/cs32/demos/templates/librarytools.cpp`
- Complete STL documentation available online at http://www.cplusplus.com/reference/stl/ and http://www.sgi.com/tech/stl/ and elsewhere

Starting Reading #7

(Notice how the two course streams have met!)

Libraries

- What is a library?
  - A compiled, packaged collection of often-used code
- Why libraries?
  - Convenient – already compiled; use again and again
  - Often allow for hardware/system-independent programming – i.e., simpler and more "portable" code
- Examples galore: C and C++ standard libraries, plus STL, graphics libraries, …
- Sometimes want to create your own libraries
  - Package together functions, related classes, class hierarchies, templates – all ready for later use

Making a library

- `ar` – Unix command to create an "archive" file
  - Mostly works like `tar` – to manage a package of files

```
% ls *.o
    tool1.o   tool2.o   tool3.o
% ar q libtools.a *.o       /* add all .o files to archive - quickly */
[% ranlib libtools.a]       /* necessary for Berkeley Unix only */
```

- Now just link a program to the library (in '.'):

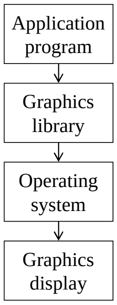
```
% g++ -Wall -o mypgm mypgm.c –ltools –L.
```

- Add/replace objects: `ar r libtools.a xx/tool4.o`
- Just read archive table of contents and other info:

```
% ar tv libtools.a
```

Graphics libraries

- OOP idea: encapsulate calls to graphics (hardware) devices
  - Provide a common interface – for using graphics on a wide variety of systems and devices
- What's the alternative?
  - Calling system and device driver-specific routines
  - Not simple, and not portable



Curses library

- Very basic graphics library to control the display of characters on a terminal screen
  - Not what most people call graphics, but cool
  - Without it, can only "print" to screen line by line
- Source must: `#include <curses.h>`
- Tell g++/gcc to link: `-lncurses`
- Then uses curses functions to open a window, and show *any character anywhere* inside it
  - e.g., `~mikec/cs32/demos/curses/rogue5.4.4`

Animating graphics

- Basic idea: move a drawing around screen
- Three essential steps to dynamic graphics – repeated over and over again in order
  1. Erase (or draw "blank" over) current drawing
  2. Move to new, nearby location, and redraw (making sure drawing happens by flushing the buffer)
  3. Pause ("sleep") so user can see drawing
  - Then go back to step 1 … and continue forever, or until animation is completed
- Speed of the animation is controlled by how long step 3 lasts – can vary for various parts

Fourth Exam
Friday, June 7