C++ STL:

Standard Template Library

— Why "Template"?

- generic programming is a strongly-typed language

- case in point:

linked list that stores arbitrary objects (e.g. void*)

- C++ discourages void * usage and requests that users of linked list specify the type

- the C++ linked list data type is:

```
std::list<...>
```

↑ Your type goes here

- we can use built-in types:

```
std::list<int> ilist;
std::list<double> dlist;
```

or we can use our user-defined type data_t:

```
std::list<data_t> list;
```

DANGER! copying objects into list very inefficient

- adding to list:

```
int num;
std::string nam;

while (!std::cin.eof()) {
    std::cin >> nam >> num;
    if (std::cin.good())
list.push_back(data_t(nam, num));
}
```

list.push_back(data_t(num, num));

- What is happening here?

  - constructor called to create new instance of data_t

  - that is passed-by-ref to push_back() which then calls copy constructor to add object to list

- if object is "large" (calls new to allocate memory) then we could spend a lot of time in a "deep copy"

- so... do what we do in Objective-C: store pointers on list

```
int num;
std::string nam;
std::list<data_t*> list;
data_t *data = NULL;

while (!std::cin.eof()) {
  std::cin >> nam >> num;
  if (std::cin.good())
list.push_back(
  }     new data_t(nam, num));
```

new data_t (nums, num);

- returns a pointer to a new object
- one "inconvenience":
  you have to remember to delete the new obj.
  (like release in Obj-C)
- benefit: fast copies onto list

- now, how to iterate thru list:
- use iterator
- like Enumerator in Obj-C

```
std::list<data_t *>::
        iterator  liti;
```

↑ the iterator

```
for (litr = list.begin();
     litr != list.end();
     litr++)
{
  std::cout << *litr;
}
```

litr is a pointer, must dereference (result is data_t * ptr)

- how to delete the list?
- remember that each object must be deleted (to free memory)

```
for (litr = list.begin();
    litr != list.end();
    litr = list.erase(litr))
{
    if(*litr) delete *litr;
}
```

instead of litr++

free memory

- there is also

```
list.clear()
```

which is supposed to do the same thing but you never know...

- other similar "containers":

```
std::vector<>
std::map<>
std::deque<>
:
```

std::vector<> - like an array, has operator[] so you can access i'th element

std::list<> - list, has no [] index

std::deque<> - double-ended queue

std::set<>

std::map<>

std::bitset<>

associative containers

(usually store value, key pairs; there's a std::pair for this

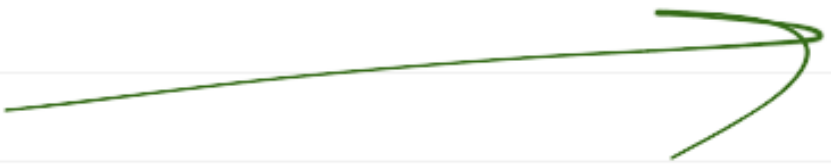
- the <> are templates, not protocol

- you can write your own templated classes, e.g.,

```
#ifndef ARRAY_H
#define ARRAY_H

//forward declaration
template <typename T>
    class array_t;
```

```
template <typename T>
class array_t {
  public:
  //constructors
   array_t (int isz = 5);
  // destructor
~array_t() { delete [] arr; }
  //friends
  // assignment operator
```



```
// member functions
int size(void) { return sz; }

T min(); // find min
T max(); // find max

// data members
private:
int sz;
T *ar;
};
#endif
```

then in array.cpp:

```
#include <array.h>
   :
   :

template <typename T>
T array_t<T>::min()
{
    T min = arr[0];

  for(int i=0; i<sz; i++)
      if (arr[i]<min) min=arr[i];
  return min;
}
```

// specializations

template class array_t<int>;

template class array_t<double>;

— forward declaration and specializations tell compiler which types to expect —
if just stubs them in for T

- what's on the horizon:

C++ 11

- redoing whole language to avoid deep copies in calls such as list.push_back()
- move semantics will now move object onto list instead of copying it

- What's the "best" language?
- it doesn't matter so long as you know how best to use it
- some concerns in all:
  - efficiency
  - memory management
  - data structure use (list, stack, queue, tree, ...)

→ CPSC 212