C++ Template Class Definitions

We examine how some of the features of C++ template class definitions work and why they are useful, by looking at some of the classes defined in STL.

- `template <typename T1, typename T2> struct pair;`
- `template <typename T> struct less;`
- `template <typename T> class vector;`

STL Pair Struct

```
template <typename T1, typename T2>
struct pair {
  T1 first;
  T2 second;

 // Constructors:
  pair(const T1& a, const T2& b)
    : first(a), second(b) { }
  pair() : first(T1()), second(T2()) { }
};
```

```
// Pair comparison operators

template <typename T1, typename T2>
inline bool operator==(const pair<T1, T2>& x,
                       const pair<T1, T2>& y) {
  return x.first == y.first && x.second == y.second;
}

template <typename T1, typename T2>
inline bool operator<(const pair<T1, T2>& x,
                      const pair<T1, T2>& y) {
  return x.first < y.first ||
         (!(y.first < x.first) && x.second < y.second);
}
```

Other Comparison Operators (in <functional>)

```
template <typename T>
inline bool operator!=(const T& x, const T& y)
  { return !(x == y); }
template <typename T>
inline bool operator>(const T& x, const T& y)
  { return y < x; }
template <typename T>
inline bool operator<=(const T& x, const T& y)
  { return !(y < x); }
template <typename T>
inline bool operator>=(const T& x, const T& y)
  { return !(x < y); }
```

STL Base Function Structs (in <functional>)

```
template <typename Arg, typename Result>
struct unary_function {
  typedef Arg argument_type;
  typedef Result result_type;
};

template <typename Arg1, typename Arg2, typename Result>
struct binary_function {
  typedef Arg1 first_argument_type;
  typedef Arg2 second_argument_type;
  typedef Result result_type;
};
```

STL Less and Greater Structs (in <functional>)

```
template <typename T>
struct less : public binary_function<T, T, bool> {
  bool operator()(const T& x, const T& y) const
  {
    return x < y;
  }
};
template <typename T>
struct greater : public binary_function<T, T, bool> {
  bool operator()(const T& x, const T& y) const
  { return x > y; }
};
```

STL Vector Class

```
// Extracted from STL vector header (old HP implementation)
// and simplified:

template <typename T>
class vector {
 public:

 // Type definitions:

  typedef T value_type;
  typedef T* iterator;
  typedef const T* const_iterator;
  typedef T& reference;
  typedef const T& const_reference;
  typedef long size_type;
  typedef ptrdiff_t difference_type;

 protected:
```

```
  iterator start;
  iterator finish;
  iterator end_of_storage;

public:
// Accessors:

  iterator begin() { return start; }
  const_iterator begin() const { return start; }
  iterator end() { return finish; }
  const_iterator end() const { return finish; }
  size_type size() const
    { return size_type(end() - begin()); }
  bool empty() const { return begin() == end(); }
  reference operator[](size_type n)
    { return *(begin() + n); }
  const_reference operator[](size_type n) const
    { return *(begin() + n); }
  reference front() { return *begin(); }
  const_reference front() const { return *begin(); }
  reference back() { return *(end() - 1); }
  const_reference back() const { return *(end() - 1); }
```

```
// Constructors, destructor, and assignment operator:

 vector() : start(0), finish(0), end_of_storage(0) {}
 vector(size_type n, const T& value = T());
 vector(const vector<T>& x);
 template <typename InputIterator>
 vector(InputIterator first, InputIterator last);
 ~vector();
 vector<T>& operator=(const vector<T>& x);

// Inserters:

 void push_back(const T& x);
 iterator insert(iterator position, const T& x);
 template <typename InputIterator>
 void insert(iterator position, InputIterator first,
             InputIterator last);
 void insert(iterator position, size_type n, const T& x);
```

```
 // Deleters:

  void pop_back();
  void erase(iterator position);
  void erase(iterator first, iterator last);
};

// Vector Comparisons

template <typename T>
inline bool operator==(const vector<T>& x,
                       const vector<T>& y) {
  return x.size() == y.size()
      && equal(x.begin(), x.end(), y.begin());
}
template <typename T>
inline bool operator<(const vector<T>& x,
                      const vector<T>& y) {
  return lexicographical_compare(x.begin(),
                      x.end(), y.begin(), y.end());
}
```