Templates

- Mechanism in C++ for generic programming
  - Also called parameterized types
- Allows programs to be parameterized on types as well as values
- Captures repeated patterns where types involved may change
- Examples:
  - Data structures – lists, vectors, arrays of different types are likely to be identical except for the element type
  - A mapping between two type values is a generic programming concept, e.g., a map between int and string, between string and string, etc
  - Even a string is a generic concept – could have other types of “character”
- Templates are recipes for code generation
- Avoids manual duplication of code and type substitution, which is very error prone and nearly impossible to maintain
- Early approaches used preprocessor macros to do systematic code duplication
  - No compiler knowledge of what was going on
  - Difficult to debug, both at the compile level and at runtime
- Functions can be parameterized on types
  - E.g., sorting algorithm is same regardless of type being sorted
- Classes can be parameterized on types
  - E.g., Data containers like vector, set, array, etc.
- Very useful for library code
  - Promotes reuse
  - Library templates should be carefully designed
  - Can force pre-instantiations for common types
- Java generics introduced in Java 1.5 are similar – based on the C++ model
  - But implemented differently by the compiler

Syntax of C++ Templates

- Keyword template is followed by one or more type names in angle brackets
  - Type names are only placeholders for concrete type names
  - May also parameterize on constant values
    - E.g., constant int sizes
- Used for parameterizing a block of code by a type name placeholder
  - Parameterization applies for the block that follows immediately
  - Block is at high level
    - class definition
    - function definition
  - Code is only partially compiled
    - Syntax is checked – very limited semantic checking
    - Type checking up to the parameterized type
    - The parameterized type can only be treated as syntactically a type about which nothing is known
  - No actual executable code is generated
    - Compiler “remembers” this template for use in instantiating code for a particular type when needed
- Instantiation of code occurs when concrete types are supplied for the parameterized block
  o E.g., for a parameterized function definition, when the function is called with concrete values, their types are used to generate a definition of the function for those types
  o E.g., for a parameterized class definition, the class name followed by concrete type names in angle brackets is an explicit reference to the concrete type of that class for those types

Examples of Template function definitions
- A template swap function that interchanges two values
  
  ```cpp
  // Swap two objects of the same type
  template <class T>
  void _swap(T & x, T & y) {
    T tmp = x; x = y; y = tmp;
  }
  ```

  - Note that swap only requires that the parameterized type T support assignment

- Template function _sort is recursive implementation of K&Rs quick sort

  ```cpp
  template <class T>
  void _sort(T & vlist, int left, int right) {
    int last;
    if (left >= right)
      return;  // Nothing to do for fewer than two elements
    _swap(vlist[left], vlist[(left + right)/2]);
    last = left;
    for (int i = left + 1; i <= right; ++i)
      if (vlist[i] < vlist[left])
        _swap(vlist[++last], vlist[i]);
    _sort(vlist[left], vlist[last]);
    _sort(vlist, last + 1, right);
  }
  ```

  ```cpp
  template <class T>
  void sort(T & vlist, int length) {
    _sort(vlist, 0, length - 1);
  }
  ```

  - Note that the only requirement is that the type T support an array operator, and the ‘elements’ support assignment and comparison with the operator ‘<’
  o These requirements cannot be enforced when just the template code is examined – they are enforced when the sort function is used for specific values. Based on the type of the value, the compiler can determine if there is assignment and comparison and thus produce compile errors if necessary. But these compile errors refer to the line where sort is called, not the line where comparison is used in the sort definition.

Function Instantiation
- Template function is not instantiated until there is a need for it
  o A call to the function requires instantiation with types corresponding to the argument types of the call
  
  - E.g.,

    ```cpp
    int ivals[] = { -9, 8, 10, 39, -13, 22, 8, 18, 44, 100, 0 };
    ```
const int ilen = sizeof(ivals)/sizeof(int);
sort(ivals, ilen);
  - Causes instantiation of sort with type T of array of int
  - In turn causes instantiation of _sort for array of int
  - And instantiation of _swap for type int

- We get separate array of string versions from:
  - string svals[:] = ...
sort(svals, slen);
- And vector<int> versions from
  - vector<int> v(ilen);
  . . .
sort(v, v.size());

Class Template Example
- Class Set is a generic container for objects of the same type
- Checks for duplicates
  - Requires comparison (overloaded ==) for objects
- Requires output method for objects
- Note that copy constructor and assignment must give the class name as parameterized – only the constructor and destructor names themselves do not need to be parameterized

```cpp
template <class TYPE>
class Set {
  public:
    // Constructor and destructor
    Set() : head(NULL) { }
    ~Set() { release(); }

    // Copy constructor
    Set(const Set<TYPE> & s) { copy(s); }

    // Assignment of sets
    Set<TYPE> & operator = (const Set<TYPE> & s) {
      if (this != & s) { release(); copy(s); }
      return *this;
    }

    // Check if a value is in the set
    bool contains(const TYPE & v) {
      for (Item * cur = head; cur != NULL; cur = cur->next)
        if (cur->value == v) return true;
      return false;
    }

    // Add the value to set if not already there
    void add(const TYPE & v) {
      if (contains(v)) return;
      Item * newitem = new Item(v);
      if (head != NULL) newitem->next = head;
      head = newitem;
    }

    // Remove the value if it's there
    void remove(const TYPE & v) {
      for (Item *prev = NULL, *cur = head;  cur != NULL;
          prev = cur, cur = cur->next) {
        if (cur->value == v) {
          Item * tmp = cur->next;
          delete cur;
          cur = tmp;
        }
      }
    }
  }
```
if (prev == NULL) head = tmp;
else prev->next = tmp;
return;
}

// Display the set
ostream & print(ostream & o) const {
for (Item * cur = head; cur != NULL; cur = cur->next) {
if (cur != head) o << ' ';
o << cur->value;
}
return o;
}

private:
// Private structure for value and linkage
struct Item {
TYPE value;
Item * next;
Item(const TYPE & v, Item * n = NULL) : value(v), next(n) { }
};

Item * head;

// Delete all items in set - used by destructor and assignment
void release() { 
Item * cur = head;
while (cur != NULL) {
Item * tmp = cur;
cur = cur->next;
delete tmp;
}
}

// Copy the set - used by copy constructor and assignment
void copy(const Set<TYPE> & s) {
Item * cur = NULL;
for (Item * exist = s.head; exist != NULL; exist = exist->next) {
Item * newitem = new Item(exist->value);
if (cur == NULL) head = cur = newitem;
else {
cur->next = newitem;
cur = cur->next;
}
}
}

};
template <class TYPE>
inline
ostream & operator << (ostream & o, const Set<TYPE> & s) { 
return s.print(o);
}

Standard Template Library
The C language has the standard C library which consists of dozens of functions that provided an interface to the operating system: system calls like open, read, write, exit, signal, etc. The library also has more functions that may provide utilities or an extra layer.
over system calls, e.g., printf, fopen (for buffered file handling), strcpy (and other functions for C string manipulation), regcmp (for regular expressions), math functions, etc. These functions may be called from C++ programs as well.

For C++, we also have the Standard Template Library (STL). Although this is not a huge collection of classes like the Java libraries, there are some basic useful classes and functions in the STL. As the name suggests, most everything in STL is template-ized, i.e., is generic with respect to types. Among the useful classes in the STL are:

- Input and output
  - Operators << and >> are defined for primitive types and may be defined for any type
  - I/O uses a notion of generic streams, which could be associated with console I/O (i.e., cin, cout, cerr), or could be streams associated with an open file or communications port, or even a string

- Container classes (sequenced and associative):
  - **vector** – a dynamic array
  - **deque** – a double ended queue
  - **list** – a doubly linked list
  - **set** – collection with own sorting and no duplicates
  - **multiset** – a set allowing duplicates
  - **map, multimap** – collection of key/value pairs
  - Containers work for any type, but are homogeneous for the type

- Containers use **iterators** to traverse the elements
  - Iterators are generic nested classes
  - Initialized with begin(), end(), rbegin(), rend()
  - Operator ++ increments to point to “next” element
  - Operator * dereferences to get the element value
  - Iterators can be defined in various flavors: bi-directional, random access, constant

- Some containers are specialized with adapters, e.g., stacks and priority queues

- Algorithms are implemented in the STL as generic functions for sorting, searching, counting, comparing, copying, swapping

- Strings are implemented in the STL as a generic string class
  - Strings can be internationalized using 16 bit characters (wstring)
  - Strings have a rich set of operators for character access, comparison, proper copying, concatenation, substrings, casting and conversions, I/O
  - Strings have many useful methods for string manipulation: regular expressions, indexing of substrings, etc.

### Example using STL vector

- Some common operations (these apply to all container classes)
  - size(), empty(), swap(), ==, !=, <

- Some operations (these apply to all container classes)
  - [] (all but list)
  - front(), back()
  - push_back(), pop_back(), (all)
  - push_front(), pop_front() (list and deque only)
```
#include <vector>

void print(const vector<int> & v) {
    cout << "[ ";
    for (vector<int>::const_iterator iter = v.begin();
        iter != v.end();
        ++iter)
        cout << *iter << " ";
    cout << "]" << endl;
}

int main() {
    vector<int> v;
    for (int i = 0; i < 10; ++i)
        v.push_back(i*i*i);
    cout << "v.size()= " << v.size() << endl;
    print(v);

    cout << "[ ";
    for (int i = v.size() - 1; i >= 0; --i)
        cout << v[i] << " ";
    cout << "]" << endl;

    v[5] = 99;
    print(v);

    return 0;
}
```

- **Other iterators** – reverse_iterator, const_iterator