Templates
- Mechanism in C++ for generic programming
- Allows programs to be parameterized on types as well as values
- Captures repeated patterns where types involved may change
- Examples:
  o Data structures – lists, vectors, arrays of different types are likely to be identical except for the element type
  o A mapping between two type values is a generic programming concept, e.g., a map between int and string, between string and string, etc
  o 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
  o No compiler knowledge of what was going on
  o Difficult to debug
- Functions can be parameterized on types
  o E.g., sorting algorithm is same regardless of type being sorted
- Classes can be parameterized on types
  o E.g., Data containers like vector, set, array, etc.
- Very useful for library code
  o Promotes reuse
  o Library templates should be carefully designed
  o Can force pre-instantiations for common types

Syntax of C++ Templates
- Keyword `template` is followed by one or more type names in angle brackets
  o Type names are only placeholders for concrete type names
  o May also parameterize on constant values
    - E.g., constant int sizes
- Used for parameterizing a block of code for a type name placeholder
  o Parameterization applies for immediately following block
  o Block is at high level
    - class definition
    - function definition
  o Code is only partially compiled
    - Syntax is checked
    - Type checking up to the parameterized type
    - The parameterized type can only be treated as syntactically a type about which nothing is known
  o No actual executable code is generated
    - Compiler “remembers” this template for use in instantiating code for a particular type
- Instantiation of code occurs when concrete types are supplied for the parameterized block
- E.g., for a parameterized function definition, when the function is called for concrete values, their types are used to generate a definition of the function for those types
- 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
  ```
  // 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
  ```
  template <class T>
  void _sort(T & vlist, int left, int right) {
    int last;
    if (left >= right)  // Nothing to do for fewer than two elements
      return;
    _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, last - 1);
    _sort(vlist, last + 1, right);
  }
  ```
- Template function sort is recursive implementation of K&Rs quick sort
  ```
  template <class T>
  void sort(T & vlist, int length) {
    _sort(vlist, 0, length - 1);
  }
  ```
- Note that only requirement is that the type T support an array operator, and the elements support assignment and comparison with ‘<’

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

int main() {
    Set<int> s;
    s.print(cout);
    return 0;
}

};

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