**Topic: Templates**

- Reusable patterns for functions and classes
- Accept data types and data items as parameters
- A way to build generic tools
- Examples:
  - universal *swap* function template
  - collections of ... whatever
  - the Standard Template Library

**Function overloading**

```c
void swap(int& a, int & b)
{
    int old_a = a;
    a = b;
    b = old_a;
}

void swap(float& a, float & b)
{
    float old_a = a;
    a = b;
    b = old_a;
}  
```

[01 Streams \ testswap.cpp]
A function template

template <class parmtyp>
void swap(parmtyp& a, parmtyp& b) 
{
    parmtyp old_a = a;
    a = b;
    b = old_a;
}

- Compiler generates no code here.

[swaptplt.cpp]

Using \textit{swap} template

int I1 = 2, I2 = 4;
float F1 = 0.5, F2 = 0.8;

swap(I1,I2); \hspace{1cm} \textit{Creates and calls integer version of swap because parameters are \texttt{ints}}
swap(F1,F2);

cout << I1 << " " << I2 << endl;
cout << F1 << " " << F2 << endl;

\begin{tabular}{|c|c|}
\hline
Output: & 4 & 2 \\
        & 0.8 & 0.5 \\
\hline
\end{tabular}

- Compiler generates code for two functions here.
Function template syntax

- Prefix to header of function template:
  ```cpp
  template <class x>
  where the identifier `x` will be used in place of some data type throughout the function template’s definition.
  ```
- Multiple type parameters may be used.
- Keyword `class` denotes a type, not necessarily a class

Class templates implement a *kind-of* relationship

- A collection of integers is a kind of collection
- A linked-list node that stores a `widget` is a kind of list node
A collection class template

```cpp
const int MAX_ITEMS = 100;

template <class T>
class collections
{
public:
    collections() { num_items = 0; }
    void insert(T new_item)
    { item[num_items++] = new_item; }
    void display()
    { for (int i=0; i < num_items; ++i)
        { cout << item[i] << " ";
            cout << endl;
        }
    }
private:
    T item[MAX_ITEMS];
    int num_items;
};
```

Code below replaces \( T \) with \( \text{employees} \)

Class is instantiated when object is declared:  
```cpp```
collections<employees> roster;
```

To use a template, instantiate it

With a parameterized type, a data type is supplied as a kind of parameter

```cpp```
collections<int>
int_colx;
int_colx.insert(326);
int_colx.insert(71);
```cpp```

Output:

```
326
71
```
Using a stack template

When using class template, component type goes here; this is a stack of ints

```cpp
stack<int> my_stack;
my_stack.push(2);
my_stack.push(4);
my_stack.push(1);
my_stack.push(7);
my_stack.push(5);
while (! my_stack.is_empty())
    cout << my_stack.pop();
```

A list-node class template

```
template <class T> class list_nodes
{
 public:
    list_nodes() { next = NULL; }
    list_nodes(T initval)
        { data = initval; next = NULL; } 
    void display() { cout << data; }
    T get_data() { return data; }
    list_nodes* get_next() { return next; }
    void set_next(list_nodes* p) { next = p; }

 private:
    T data;
    list_nodes* next;
}
```

This list-node class accommodates data of any type (T); a corresponding list class would have insertion, deletion member functions
A list-based stack template

```cpp
template <class T>
class stacks {
    public:
        stacks() { first = NULL; }
        void push(T new_item);
        T pop();
        bool is_empty()
            { return (first == NULL); }
    private:
        list_nodes<T>* first;
};
```

- A list node has a `T` as a member; stack declared with `stacks<int>`, e.g., is of `int`

A template may have data parameters

```cpp
template <class T, int max_sz>
class collections {
    T* ele;
    int curr_size;
    public:
        collections()
            { curr_size = 0; ele = new T[max_sz]; }
};
```

**Usage:** `collections<int,10> collec;` (constructs 10 element collection of int)
STL implementation of collections, iterators, algorithms

- Tools used:
  - Templates
  - Operator functions
- For performance reasons, weak use is made of inheritance
- Inline functions
- Encapsulation

Implementing some STL-style templates

```cpp
template<class T>
class vectors {
    ...  
    T item[MAX_ITEMS];
    int size;
};

template <class Tcollection,class Telement>
class iterators {
    ...  
    Tcollection* p_collection;
    Telement* p_element;
};

// Generic search algorithm:
template<class I,class T>
I find(I first,I last,T key) {
    I i = first;
    while (i != last && *i != key)
        ++i;
    return i;
} // stldemo.h
```
Advantages of templates

- Code reuse
- Type-independent building blocks
- Abstraction of type of contained object from other features of container
- Abstraction of container size
- Code maintenance
- Make possible parameterized types for generic programming