Type independence

- Many algorithms like search, sort, or swap do not depend on the type of the elements/items.
- We would like to re-use the same code regardless of the item type...
- without having to maintain duplicate copies:
  - sortIntArray (int a[]; int numValues)
  - sortFloatArray (float a[]; int numValues)
  - sortCharArray (char a[]; int numValues)

Generic programming

- Writing functions and classes that are type-independent is called generic programming.
- These functions and classes will have an extra parameter to represent the specific type of the components.
- When the function is called, or class is instantiated, the programmer provides the specific type:
  ```cpp
  vector<string> students (20);
  vector<double> dailySales (365);
  ```

Templates

- C++ provides templates to implement generic functions and classes.
- A function template is not a function, it is a design or pattern for a function.
- The function template makes a function when the compiler encounters a call to the function.
  - Like a macro, it substitutes appropriate type
Example function template

```cpp
swap
template <class Object>
void swap (Object &lhs, Object &rhs) {
    Object tmp = lhs;
    lhs = rhs;
    rhs = tmp;
}
```

```cpp
int main() {
    int x = 5;
    int y = 7;
    string a = "hello";
    string b = "there";
    swap <int> (x, y);    //int replaces Object
    swap <string> (a, b); //string replaces Object
    cout << x << "  " << y << endl;
    cout << a << "  " << b << endl;
}
```

```
7  5
there  hello
```

Notes about the example

- Normal syntax to call the templated function include the type: <int>
  - swap<int> (x,y);
  - swap<string> (a,b);
- It’s not necessary to specify the type when the compiler is capable of figuring it out from context.
  - swap (x,y);
  - swap (a,b);

Notes about the example

- The header: template <class Object>
  - `class` is a keyword. You could also use `typename`:
    template <typename Object>
- Object is the parameter name. You can call it whatever you like.
  - it is often capitalized (because it is a type)
  - names like T and U are often used
- The parameter name (Object in this case) can be replaced ONLY by a type.

How function templates work

- The compiler will not use the pattern unless/until it encounters a call to the function.
  - At that point, the compiler performs the text substitution you asked for, and then compiles the newly generated function as if you’d written that function yourself.
- What happens if I instantiate the same template multiple different ways?
  - It is just function overloading, you get two or more functions with the same name, but with different arguments!
Class Templates

- Template classes work similarly to template functions with the following exceptions:
  - The compiler will never guess at type argument for a template class, you must always use `<...>`.
  - Classes cannot be “overloaded”, but the compiler will permit you to instantiate the same template class in multiple ways.
    - Each distinct instantiation results in a completely distinct class! (with its own copy of the static data members, for example).
  - The member functions in a template class are template functions (requiring the header).

Simple example, class template MemoryCell (formerly IntCell)

```cpp
#include <iostream>
using namespace std;

int main() {
    MemoryCell<int> m;
    m.write(5);
    cout << "Cell contents are " << m.read() << endl;
}
```

Output:

```
Cell contents are 5
```

Simple example, class template MemoryCell

```cpp
// Object: must have zero-parameter constructor and operator=
template <class Object>
class MemoryCell {
    public:
        // Construct a MemoryCell.
        explicit MemoryCell ( const Object & initVal = Object () )
            : storedValue (initVal) { }

        // public methods
        Object read () { return storedValue; }
        void write (Object x) { storedValue = x; }

    private:
        Object storedValue; // stores the memory cell contents
};
```

Example 2, class template vector: class decl

```cpp
// A barebones vector ADT
// T: must have zero-parameter constructor and operator=
template <typename T>
class vector {
    public:
        explicit vector(int initial_capacity=8);
        void push_back(T);    // number of elements in vector
        T pop_back();
        T operator[](int k);
    private:
        T* data; // stores data in dynamically allocated array
        int length;
        int capacity; // size of array, to know when to expand
        void expand(void); // to increase capacity as needed
};
```
**Example 2, class template**

vector, function definitions

template <typename T>
vector<T>::vector(int init_cap) {
  capacity = init_cap;
  data = new T[capacity];
  length = 0;
}
template <typename T>
void vector<T>::push_back(T x) {
  if (capacity == length)
    expand();
  data[length] = x;
  length ++;
}
template <typename T>
T vector<T>::pop_back() {
  if (length <= 0) throw "AttemptToPopFromEmptyVectorException";
  length--;
  return data[length];
}

```
```

**Example 2, class template**

vector, function definitions

template <typename T>
T vector<T>::operator[](int k) {
  if (k<0 || k>=length)
    throw "ArrayIndexOutOfBoundsException";
  return data[k];
}
template <typename T>
void vector<T>::expand(void) {
  capacity *= 2;
  T* new_data = new T[capacity];
  for (int k = 0; k < length; k += 1)
    new_data[k] = data[k];
  delete[] data;
  data = new_data;
}

**Simple example, class template**

MemoryCell

int main() {
  vector<string> m(2);
  m.push_back("As");
  m.push_back("Ks");
  m.push_back("Qs");
  m.push_back("Js");
  for (int i=0; i<4; i++) {
    cout << m[i] << endl;
  }
}

Output:

| As | Ks | Qs | Js |

```

Could have used pop_back, it works too.
```

**Class Templates and .h files**

- Template classes cannot be compiled separately
  - Machine code is generated for a template class only when the class is instantiated (used).
  - If you compile a template (class declarations + functions definitions) it will not generate machine code.
  - When a file using (instantiating) a template class is compiled, it requires the complete definition of the template, including the function definitions.
  - Therefore, for a class template, the class declaration AND functions definitions must go in the header file.
  - It is still good practice to define the functions outside of (after) the class declaration.