Introduction to Templates and The STL

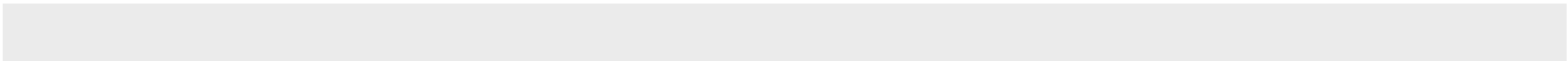
See also: Chapter 4 (89-100), Chapter 11 (279-281), and Chapter 21

```
// GameBoard.h

class GameBoard
{
    public:
        // The general-purpose GameBoard allows the user to specify its dimensions
        GameBoard(int inWidth, int inHeight);
        GameBoard(const GameBoard& src); // Copy constructor
        ~GameBoard();
        GameBoard &operator=(const GameBoard& rhs); // Assignment operator

        void setPieceAt(int x, int y, const GamePiece& inPiece);
        GamePiece& getPieceAt(int x, int y);
        const GamePiece& getPieceAt(int x, int y) const;

        int getHeight() const { return mHeight; }
        int getWidth() const { return mWidth; }

    protected:
        void copyFrom(const GameBoard& src);
        // Objects dynamically allocate space for the game pieces.
        GamePiece** mCells;
        int mWidth, mHeight;
```

```
};
```

Templates

Strengths	Weaknesses
Type Safe	"Code Bloat"
Compile-Time Checks	Gross Syntax
Easy to Use	Always Homogenous



```cpp
// Grid.h

template <typename T>
class Grid
{
    public:
        Grid(int inWidth, int inHeight);
        Grid(const Grid<T>& src);
        ~Grid();
        Grid<T>& operator=(const Grid<T>& rhs);

        void setElementAt(int x, int y, const T& inElem);
        T& getElementAt(int x, int y);
        const T& getElementAt(int x, int y) const;
        int getHeight() const { return mHeight; }
        int getWidth() const { return mWidth; }
        static const int kDefaultWidth = 10;
        static const int kDefaultHeight = 10;

    protected:
        void copyFrom(const Grid<T>& src);
        T** mCells;
        int mWidth, mHeight;
};
```

```
Grid<int> myIntGrid(5, 5); // Declares a 5x5 grid that stores ints
myIntGrid.setElementAt(0, 0, 10);
int x = myIntGrid.getElementAt(0, 0);

Grid<int> grid2(myIntGrid);
```

```
Grid<SpreadsheetCell> mySpreadsheet(10, 10);
SpreadsheetCell myCell;
mySpreadsheet.setElementAt(3, 4, myCell);
```

```
Grid<char*> myStringGrid(2, 2);
myStringGrid.setElementAt(2, 2, "hello");
```

```
Grid<Grid<int> > myGridOfGrids(3, 3);
myGridOfGrids.setElementAt(0, 1, myIntGrid);
```

```
myIntGrid = myStringGrid; // BUG! They're different types!
```

```
void foo(Grid<int>& inGridBergman); // The template type is part of the type
```

Selective Instantiation:

If your class doesn't support features that the template class uses, those parts of the template class will be omitted.

The Standard Template Library (STL)

Strengths	**Weaknesses**
Standardized	Steep Learning Curve
Flexible	Gross Syntax
Comprehensive	Heavyweight
Helps Avoid Dynamic Memory	Not Thread Safe
Extensible	Strange Omissions
	Often Chokes on References
	Hard to Extend

A good abstraction makes the common case easy and the rare case possible.

STL Containers

- vector
- list
- deque

- stack
- queue

- map
- multimap
- set
- multiset

STL Algorithms – *Generic way of performing a task*

STL Iterators – *Objects for accessing elements of a container.*

Each STL container has a corresponding iterator.

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

int main(int argc, char** argv) {
    vector<double> doubleVector(10, 0); // Create a vector of 10 doubles, initialized to 0
    double max;

    cout << "Enter score 1: ";
    cin >> doubleVector[0];
    max = doubleVector[0];

    for (int i = 1; i < 10; i++) {
        cout << "Enter score " << i + 1 << ": ";
        cin >> doubleVector[i];
        if (doubleVector[i] > max) {
            max = doubleVector[i];
        }
    }

    max /= 100;
    for (int i = 0; i < 10; i++) {
        doubleVector[i] /= max;
        cout << doubleVector[i] << " ";
    }
    cout << endl;
}
```

```
int main(int argc, char** argv) {
    vector<double> doubleVector; // Create a vector with zero elements.
    double max, temp;
    size_t i;

    cout << "Enter score 1: ";
    cin >> max;
    doubleVector.push_back(max);

    for (i = 1; true; i++) {
        cout << "Enter score " << i + 1 << " (-1 to stop): ";
        cin >> temp;
        if (temp == -1) {
            break;
        }
        doubleVector.push_back(temp);
        if (temp > max) {
            max = temp;
        }
    }

    max /= 100;
    for (i = 0; i < doubleVector.size(); i++) {
        doubleVector[i] /= max;
        cout << doubleVector[i] << " ";
    }
    cout << endl;
}
```

```
class Element
{
    public:
        Element() {}
        ~Element() {}
};

int main(int argc, char** argv)
{
    vector<Element> elementVector;
    Element e1;
    elementVector.push_back(e1);
}
```

Working with vectors

```
// vectors on the heap
vector<Element>*  elementVector = new vector<Element>(10);
delete elementVector;
```

```
// reusing a vector
vector<int> intVector(10, 0);
// Other code . . .
intVector.assign(5, 100);
```

```
// comparing vectors
if (vectorOne == vectorTwo) {
    cout << "equal!\n";
} else {
    cout << "not equal!\n";
}
```

See reference for other vector methods…

vector Iterators

```
for (i = 0; i < doubleVector.size(); i++) {
    doubleVector[i] /= max;
    cout << doubleVector[i] << " ";
}
```

```
for (vector<double>::iterator it = doubleVector.begin();
    it != doubleVector.end(); ++it)
{
    *it /= max;
    cout << *it << " ";
}
```

-> works with iterators:

```
for (vector<SCell>::iterator it = sheet.begin(); it != sheet.end(); it++) {
    cout << it->getValue(); << endl;
}
```

const iterators:

```
vector<type>::const_iterator it = myVector.begin();
```

Why use iterators?

They indicate positions and ranges:

```
    vector<int> vectorOne, vectorTwo;
    int i;

    vectorOne.push_back(1);
    vectorOne.push_back(2);
    vectorOne.push_back(3);
    vectorOne.push_back(5);

    // Oops, we forgot to add 4. Insert it in the correct place.
    vectorOne.insert(vectorOne.begin() + 3, 4);
```

```
    // Add elements 6 through 10 to vectorTwo.
    for (i = 6; i <= 10; i++) {
        vectorTwo.push_back(i);
    }

    // Add all the elements from vectorTwo to the end of vectorOne.
    vectorOne.insert(vectorOne.end(), vectorTwo.begin(),
                     vectorTwo.end());

    // Clear vectorTwo entirely.
    vectorTwo.clear();

    // And add 10 copies of the value 100.
    vectorTwo.insert(vectorTwo.begin(), 10, 100);

    // Decide we only want 9 elements.
    vectorTwo.pop_back();

    // Now erase the numbers 2 through 5 in vectorOne.
    vectorOne.erase(vectorOne.begin() + 1, vectorOne.begin() + 5);
```

Associative Containers

A container that stores keys and values instead of ordered elements

Examples:

- map
- multimap
- set
- multiset

The `pair` Class

```
#include <utility>
#include <string>
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    pair<string, int> myPair("hello", 5), myOtherPair;

    myOtherPair.first = "hello";
    myOtherPair.second = 6;

    pair<string, int> myThirdPair(myOtherPair);

    if (myPair < myOtherPair) {
        cout << "myPair is less than myOtherPair\n";
    } else {
        cout << "myPair is greater than or equal to myOtherPair\n";
    }

    if (myOtherPair == myThirdPair) {
        cout << "myOtherPair is equal to myThirdPair\n";
    } else {
        cout << "myOtherPair is not equal to myThirdPair\n";
    }
}
```

make_pair function

```
pair<int, int> aPair = make_pair(5, 10);
```

```
void foo(pair<int, int> inPair);
```

```
int main()
{
    pair<int, int> myPair(5, 10);
    foo(myPair);

    foo(make_pair(5, 10));   // shortcut using make_pair()
}
```

```
#include <map>
using namespace std;

class Data
{
    public:
        Data(int val) { mVal = val; }
        int getVal() const { return mVal; }
        void setVal(int val) {mVal = val; }
        // Remainder of definition omitted
    protected:
        int mVal;
};

int main(int argc, char** argv)
{
    map<int, Data> dataMap;
}
```

Forget insert(), use []

```
dataMap[1] = Data(4);
dataMap[1] = Data(6); // Replaces the element with key 1
```

Even with non-ints:

```
map<string, string> nameMap;
nameMap["Kleper"] = "Scott";
```

Map iterator:

```
for (map<int, Data>::iterator it = dataMap.begin();
    it != dataMap.end(); ++it)
{
    cout << it->second.getVal() << endl;
}
```

Getting values out of a map by key:

```
cout << "First name of Kleper is " << nameMap["Kleper"] << endl;
// Previous line will insert key "Kleper" if it doesn't exist!
```

To lookup a value without possibility of adding a new one, use find() with an iterator:

```
map<string, string>::iterator it = nameMap.find("Kleper");
if (it != dataMap.end()) {
    cout << "First name of Kleper is " << it->second;
}
```

To remove an element, use erase()

```
cout << "There are " << nameMap.count("Kleper") << " Klepers\n";
nameMap.erase("Kleper");
cout << "There are " << nameMap.count("Kleper") << " Klepers\n";
```