

Genericity Example.

```
// Collection specification

#ifndef _COLLECTION_H_
#define _COLLECTION_H_

#include <assert.h>

template<class TYPE>
class Collection
{
public:
    Collection();
    Collection(int);
    Collection(int, TYPE);
    ~Collection();

    int GetNumberOfElements();
    int GetCapacity();
    void ChangeElement(int, TYPE);
    TYPE operator [] (int);
    void operator +(TYPE);
    void operator -(int);

protected:
    int m_nAmount_Of_Elements;
    int m_nCapacity;
    TYPE *m_Elements;
};

#endif // !_COLLECTION_H_

// Collection implementation

/*
 * Default constructor.
 */
template<class TYPE>
Collection::Collection() : m_nCapacity(0), m_nAmount_Of_Elements(0)
{
    m_Elements = NULL;

    assert(m_nCapacity == 0);
    assert(m_nAmount_Of_Elements == 0);
    assert(m_Elements == NULL);
}

/*
 * Constructor, initialize to size.
 */
template<class TYPE>
```

```

Collection::Collection(int size) : m_nCapacity(size),
m_nAmount_Of_Elements(0)
{
    m_Elements = new TYPE[m_nCapacity];

    assert(m_nCapacity >= 0);
    assert(m_nAmount_Of_Elements == 0);
    assert(m_Elements != NULL);
}

/*
 * Constructor, initialize to size and parameter value.
 */
template<class TYPE>
Collection::Collection(int size, TYPE initial_value) :
m_nCapacity(size), m_nAmount_Of_Elements(size)
{
    m_Elements = new TYPE[m_nCapacity];

    for (int index = 0; index < m_nCapacity; index++)
    {
        m_Elements[index] = initial_value;
    }

    assert(m_nCapacity >= 0);
    assert(m_nAmount_Of_Elements <= m_nCapacity);
    assert(m_Elements != NULL);
}

/*
 * Destructor.
 */
template<class TYPE>
Collection::~Collection()
{
    assert(m_Elements != NULL);

    delete [] m_Elements;
    m_Elements = NULL;
}

/*
 * Return the total number of elements.
 */
template<class TYPE>
int Collection::GetNumberOfElements()
{
    assert(m_Elements != NULL);
    assert(m_nAmount_Of_Elements >= 0);

    return m_nAmount_Of_Elements;
}

/*
 * Return the collection size.
 */
template<class TYPE>

```

```

int Collection::GetCapacity()
{
    assert(m_Elements != NULL);
    assert(m_nCapacity >= 0);

    return m_nCapacity;
}

/*
 * Return an indexed element.
 */
template<class TYPE>
TYPE Collection::operator [] (int index)
{
    assert(m_Elements != NULL);
    assert(index >= 0);
    assert(index < m_nCapacity);

    return m_Elements[index];
}

/*
 * Delete an element.
 */
template<class TYPE>
void Collection::operator -(int index)
{
    assert(m_Elements != NULL);
    assert(index <= m_nAmount_Of_Elements);
    assert(m_nCapacity > 0);
    assert(m_nAmount_Of_Elements > 0);
    assert(m_nAmount_Of_Elements <= m_nCapacity);

    for (int k = index; k < m_nCapacity; k++)
    {
        m_Elements[k] = m_Elements[k + 1];
    }

    m_nAmount_Of_Elements--;

    assert(m_nAmount_Of_Elements <= m_nCapacity);
    assert(m_Elements != NULL);
}

/*
 * Add an element to the end of the collection.
 */
template<class TYPE>
void Collection::operator +(TYPE value)
{
    assert(m_Elements != NULL);
    assert(m_nAmount_Of_Elements < m_nCapacity);

    m_Elements[m_nAmount_Of_Elements++] = value;

    assert(m_nAmount_Of_Elements <= m_nCapacity);
    assert(m_Elements != NULL);
}

```

```
}

/*
 * Change an element.
 */
template<class TYPE>
void Collection::ChangeElement(int index, TYPE value)
{
    assert(m_Elements != NULL);
    assert(m_nCapacity > 0);
    assert(m_nAmount_Of_Elements > 0);
    assert(m_nAmount_Of_Elements >= index);

    m_Elements[index] = value;

    assert(m_nAmount_Of_Elements <= m_nCapacity);
    assert(m_Elements != NULL);
}
```

Additional Files

http://www.bletchleypark.net/algorithms/generic_cpp.txt

http://www.bletchleypark.net/algorithms/parameter_cpp.txt

http://www.bletchleypark.net/algorithms/parameter_h.txt