**“vvector”: Sequence container to store dynamic size objects in contiguous memory location.**

CSE 622 Generic Programming

Term Project Report

**Ghanshyam Bhutra**

Texas A&M University

UIN 323006750

gbhutra@tamu.edu

**Siddhant Rath**

Texas A&M University

UIN 124007171

rath.siddhant@tamu.edu

CONTENTS:

I. User Guide 3

A. Introduction 3

B. Overview: 3

C. Usage: 3

1) Efficient storage of UTF-8 encoded strings: 3

2) Storage of XML/JSON Data: 3

3) Storing Results of SQL Queries : 3

D. Example : 4

E. Notes: 4

II. Reference Manual 5

A. Description: 5

B. Syntax: 5

C. Model of 6

1) Random Access Container [2] 6

2) Back Insertion Sequence [3] 6

3) Refinement of Sequence[4] 6

D. Type Requirements 6

E. Members 6

F. Notes 7

III. Design Document 8

IV. Testing 8

V. Source Code 8

VI. References 12

# User Guide

## Introduction

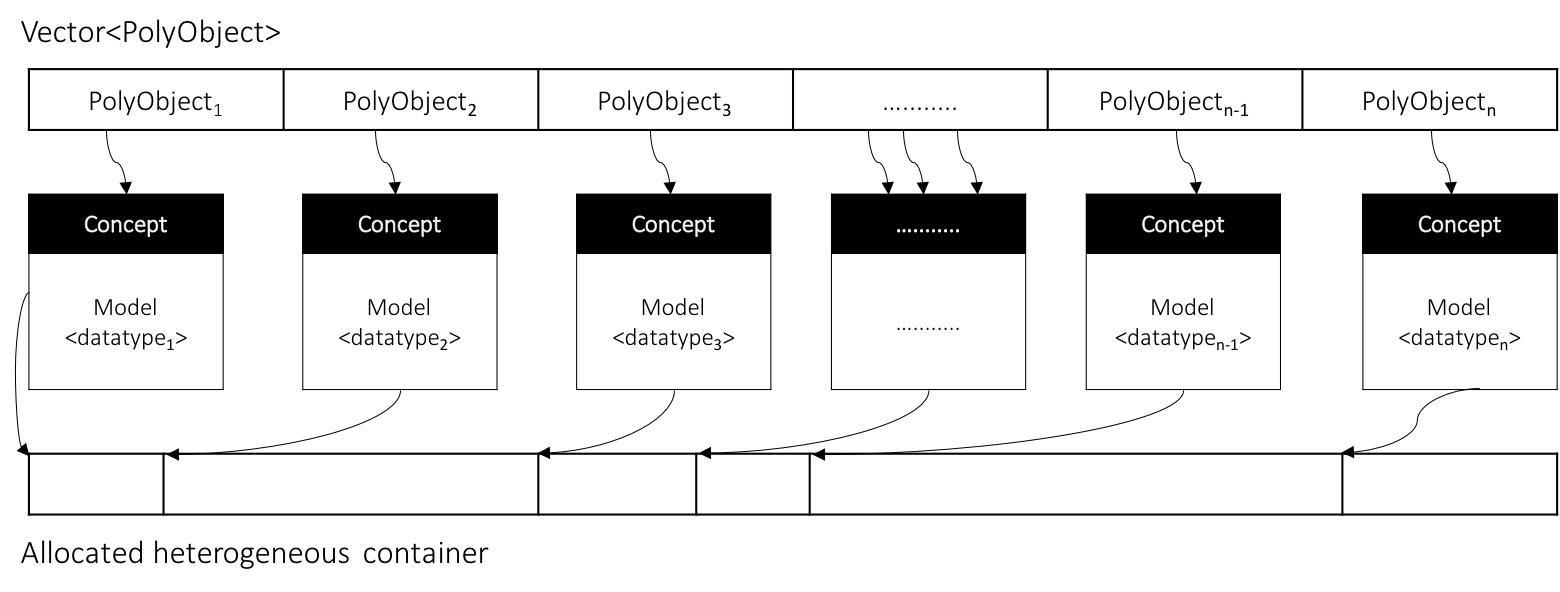
A heterogeneous container in an object that can store elements of different types. Programmers frequently come across need of such containers in a wide variety of applications. Although, in C++, many techniques can be used to simulate such a behavior its not an in-built feature. The heterogeneous containers offered in the standard template library are inherently fixed size (*std::tuple, std::set*) which limits their functionality. This document introduces a vector like heterogeneous container which is a collection of elements with different types. Henceforth referred as “*vvecto*r”, it supports standard container operations for look up, iteration, insertion, deletion and others.

This document is intended to be used by an audience with familiarity in C++ and STL.

## Overview:

vvector is a sequence container which encapsulates dynamic size collection of heterogeneous values. The elements are always stored contiguously in memory. Elements in vvector can be accessed via iterators that supports forward, bidirectional and random access iteration. Elements can be inserted or deleted at beginning or in the middle. The size of the vvector is handled internally such that once the vvector is initialized by the user, it automatically grows when new elements are added to it.

Internally vvector maintains a vector of type poly\_object\_t and a block of memory. The class poly\_object\_t has a shared pointer to an abstract class which is inherited by a templatized class which models the type of object stored in the vvector. When an object is pushed into the vvector it gets copy constructed into its memory block and the pointer to the object is modelled into poly\_object\_t and stored into the vector. The vvector provides interface to access its elements by the wrapping the iterators of the vector of poly\_object\_t.



## Usage:

The following code snippet shows some of the most common features of vvector.

#include<vvector.h>

vvector<less\_than\_comparable,ostreamable> V;

/\*

The template parameters are the concepts that must be implemented by elements within the vvector

/\*

int i1=5,i2=7;

char c=’s’;

std::string str = “I am a variable vector”;

V.push\_back(i1);

V.push\_back(i2);

if (V[1]>V[0])

V.push\_back(c);

V.push\_back(str);

for(auto v: V) {

std::cout<<”\t”<<v;

}

std::cout<<”\n”;

OUTPUT:

5 7 s I am a variable vector

The previous examples show a trivial example usage of vvector. The template parameters used in the initialization of vvector enlist the concepts supported by its elements. The current implementation of the vvector supports four combinations of such concepts the details of which are explained later. ‘vvector’ could be used in any user defined objects as long as the concepts declared in the vvector initialization is implemented by the user defined class.

The generic implementation of vvector makes it usable in a variety of applications. The following discusses some instances which could benefit from vvector.

### Efficient storage of UTF-8 encoded strings:

UTF-8 is a character encoding capable of encoding all possible character. The encoding is of variable length and each character can be 1 to 6 bytes long. Homogenous style implementation declares the highest size data type that is expected to be found in the UTF-8 string. For example, if a UTF-8 encoded text is expected to have characters that can be up to 4 bytes long, all characters are declared in a four-byte data type say ‘*wchar32\_t*’. Memory is not managed efficiently as characters of lower byte size are padded with zero. A better way to store theses strings is by detecting each character’s size from the UTF-8 encoding and storing them in exactly matching data type. A one byte UTF-8 character can be stored in a char, 2 bytes, say in *wchar* and so on. And all these sequence of different size characters needs to be stored in a container. This heterogeneous container is made possible by vvector.

### Storage of XML/JSON Data:

XML elements, JSON objects are heterogeneously typed. In fact, these kind of structures are nested collections that vary in size. To preserve the data types represented by these documents it is required to be stored in a heterogeneous structure.

# Reference Manual

**vvector**



**Category:** containers



**Component Type:** Type

## Description:

The vvector is a heterogeneous container that organizes elements of different type in a linear sequence. It provides random access to its elements, and dynamic additions and removal to and from the sequence. Just like it’s homogenous counterpart vector, vvectors use contiguous storage locations for their elements. This means that elements can be accessed using offsets on regular pointer to its elements. The size increases dynamically and it is automatically maintained by the container.

vvector use a generic concept called PolyObject. These PolyObject model different types of pointer depending on the element that is being pushed. The pointer holds the location in the memory block owned by vvector where the object is stored. The elements are constructed linearly into the memory block and their lifetime is same as the lifetime of the vvector.

## Template Parameters

The following template parameters can be used to customize the type of object intended to be stored in vvector. All combinations in any sequence is supported.

### less\_than\_comparable : It must be possible to compare two objects that are pushed into the vvector using operator <,<=, > and >=.

### ostreamable : It must be possible to stream an object that is pushed into the vvector using operator <<

### equality\_comparable : It must be possible to compare two objects that are pushed into the vvector using operator == and operator !=

### addable : The object that is pushed into the vvector must have an overload for the + operator.

## Syntax:

Vvector<ostreamable> V;

float f = 0.05f;

V.push\_back(5);

V.push\_back(‘a’);

V.push\_back(“hello”);

V.push\_back(f);

for(auto v : V)

std::cout<<v<<endl;

char k = v[1].getVal<char>(); // k is of type char

auto z = v[1]; // z is of type poly\_object\_t< >

## Refines

### Sequence[4]

A sequence is a variable sized container whose elements are arranged in a strict linear order

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Expression | Precondition | Semantics |
| Default Constructor | vvector v |  | Creates a vvector having a memory block of 200 bytes. |
| Insert | v.insert(p, element) | p is a valid iterator in v | A copy of element is inserted before p. Elements after p are shifted. |
| Erase | v.erase(p) | p is a iterator in v. | Destroys the element at p. Elements after p are shifted. |
| Clear | v.clear() |  | Clears the sequence |

### Assignable

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Expression | Precondition | Semantics |
| Copy Constructor | vvector V(x) |  |  |
| Copy Constructor | vvector V = X; |  |  |
| Assignment | V = X |  |  |
| Swap | v.swap(x) |  |  |

## Models

### Random Access Container [2]

A random access container is a reversible container whose iterator type is random access iterator. It provides amortized constant time access to arbitrary elements.

Expression semantics:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Expression | Precondition | Semantics |
| Element Access | V[n] | 0 <= n < v.size() | Returns the nth element from the beginning of the container |

### Back Insertion Sequence [3]

A back insertion sequence is a sequence where it is possible to append an element to the end, or to access the last element in amortized constant time.

Valid Expression:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Expression | Precondition | Semantics |
| Back | v.back() | !v.empty() | Equivalent to \*(--v.end()). |
| Push Back | v.push\_back(element) |  | Equivalent to v.insert(v.end(), element) |
| Pop Back | v.pop\_back() | !v.empty() | Equivalent to v.erase(--v.end()) |

## Members

|  |  |  |
| --- | --- | --- |
| **Member** | **Concept in STL** | **Descriptor** |
| iterator begin() | Container | Returns an iterator pointing to the beginning of the vector. |
| iterator end() | Container | Returns an iterator pointing to the end of the vector. |
| size\_type size() const | Container | Returns the size of the vector. |
| bool empty() const | Container | true if the vector's size is 0. |
| reference operator[](size\_type n) | Random Access Container | Returns the n'th element. |
| vvector() | Container | Creates an empty vvector. |
| vvector(const vvector&) | Container | The copy constructor. |
| ~vvector() | Container | The destructor |
| reference front() | Sequence | Returns the first element. |
| reference back() | Back Insertion Sequence | Returns the last element. |
| void push\_back(const T&) | Back Insertion Sequence | Inserts a new element at the end. |
| void pop\_back() | Back Insertion Sequence | Removes the last element. |
| iterator insert(iterator pos, const T& x) | Sequence | Inserts x before pos. |
| void insert(iterator pos, size\_type n, const T& x) | Sequence | Inserts n copies of x before pos. |
| iterator erase(iterator pos) | Sequence | Erases the element at position pos. |
| iterator erase(iterator first, iterator last) | Sequence | Erases the range [first, last) |
| void clear() | Sequence | Erases all of the elements. |
| getVal<>() | vvector |  |
| size\_t byte\_size() | vvector | Returns the number of bytes occupied by all the elements in vvector. |

**poly\_object\_t**

## Description:

The poly\_object\_t is a class which models different types of objects using a templatized child class which in turn is pointed by common base class.

## Template Parameters

Based on the templates parameters of vvector the poly\_object\_t objects are fully specialized to support the following concepts of the object type they model.

### less\_than\_comparable : It must be possible to compare two objects that are pushed into the vvector using operator <,<=, > and >=.

### ostreamable : It must be possible to stream an object that is pushed into the vvector using operator <<

### equality\_comparable : It must be possible to compare two objects that are pushed into the vvector using operator == and operator !=

### addable : The object that is pushed into the vvector must have an overload for the + operator.

# Design Document

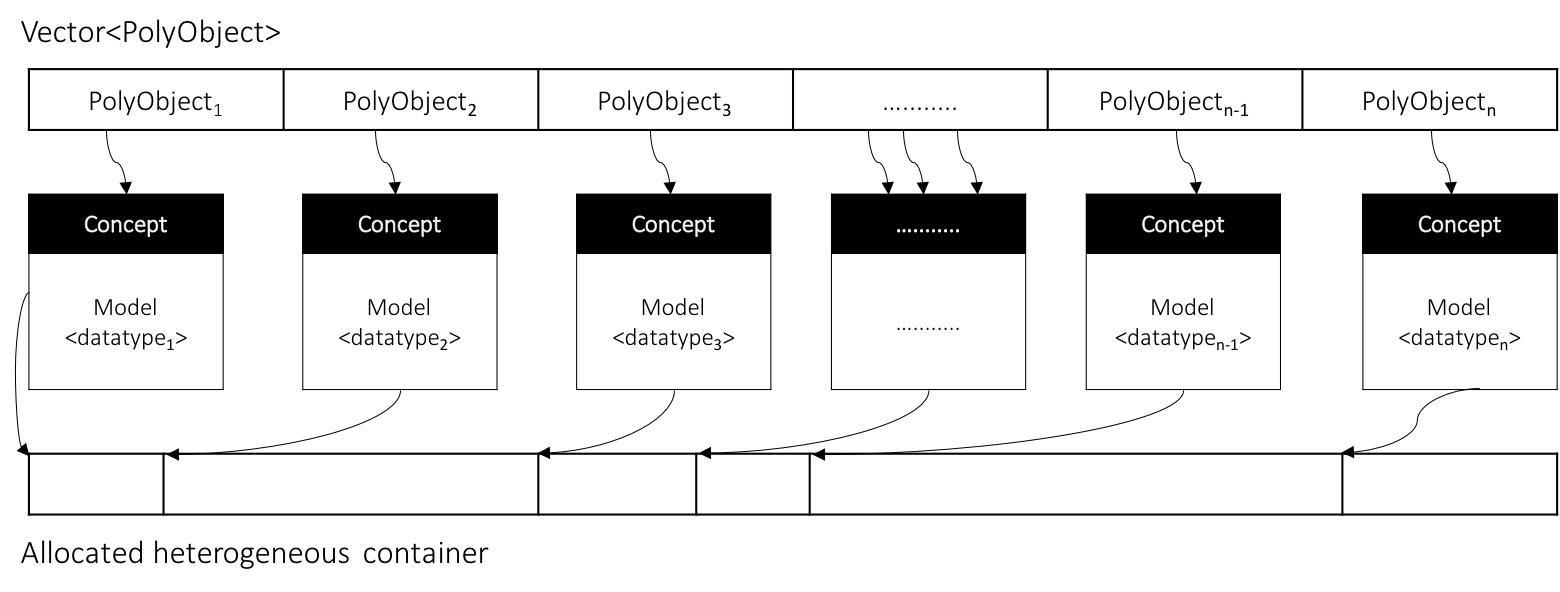
The vvector models the concept of a sequence. It’s a dynamic size container of heterogeneous objects stored in contiguous memory location. The vvector employs late binding using virtual methods, due to which the compiler resolves the references to the objects stored in the vvector at runtime. This allows the vvector to operate on objects of different kinds.

Internally vvector maintains two dynamic memory blocks to store its heterogeneous elements. The STL container vector is used as one of the memory block to store the references to the objects stored in the other block of memory which owned by vvector. A data structure poly\_object\_t is modeled as per the following class diagram. The template parameters of vvector is used to customize the type of poly\_object\_t it can hold. This feature allows generic usage of vvector.



poly\_object\_t is a class which contains a shared pointer of an abstract class concept\_t. The class *concept\_t* is inherited by a class *model\_t* which has a pointer. The member pointer of class model is instantiated with the same data type used to instantiate the class *model*. The compiler creates instances of the model class based on the different types of data pushed into the vvector. The poly\_object\_t objects accesses the pure virtual functions of the concept\_t class via a shared pointer. During runtime it gets dispatched to the functions which are defined in model\_t class. The size of an instantiation of a poly\_object\_t class is always fixed, it has only two pointers as its member variables. This makes it possible to be stored efficiently in a container like vector. Runtime polymorphism of poly\_object\_t (polymorphic object) makes it a generic data structure to create references of different kinds of objects.

The objects passed to vvector are copy constructed into a memory block owned by vvector using placement new operator. Then the pointer of the same is used to create an instance of PolyObject. The instance of the vvector is stored in the std::vector owned by vvector. Figure illustrates the mechanism used to store the heterogeneous elements in vvector.



The access to the vvector elements is provided by the std::vector<poly\_object\_t>. Using std::vector as the storage container for poly\_object\_t, it allows random access to the elements of vvector.

# Source Code

The source code is split into two files vvector.h and polyObject.h respectively. A program using vvectors would need to include only the vvector.h. Both files can be found in the zip file gp\_project.zip

# References

[1] Strongly Typed Heterogeneous Collection: Oleg Kiselyov FNMOC, Monterey, CA, Ralf La ̈mmel Keean Schupke, VU & CWI, Amsterdam Imperial College, London

[2] Random Access Container: <https://www.sgi.com/tech/stl/RandomAccessContainer.html>

[3] Back Insertion Sequence: <https://www.sgi.com/tech/stl/BackInsertionSequence.html>

[4] Sequence: <https://www.sgi.com/tech/stl/Sequence.html>