Programmer's Reference by *M Gaffiero*

gaffie@users.sourceforge.net

Class::STL::Containers

1.0

Table of Contents
Class::STL::Containers

NAME

Class::STL::Containers - Perl extension for STL-like object management

SYNOPSIS

```
use Class::STL::Containers;
use Class::STL::Algorithms;
use Class::STL::Utilities;
use Class::STL::Iterators;

# Deque container...
my $d = deque();
$d->push_back($d->factory(data => 'first'));
$d->push_back($d->factory(data => 'second'));
$d->push_back($d->factory(data => 'third'));
$d->push_back($d->factory(data => 'fourth'));
$d->push_back($d->factory(data => 'fifth'));
$d->push_front($d->factory(data => 'seventh'));
$d->pop_front(); # remove element at front.
$d->pop_back(); # remove element at back.
::foreach($d->begin(), $d->end(), MyPrint->new());

# MyPrint Unary Function -- used in ::foreach() above...
{
  package MyPrint;
  use base qw(Class::STL::Utilities::FunctionObject::UnaryFunction);
  sub function_operator
  {
    my $self = shift;
    my $arg = shift;
    print "Data:", $arg->data(), "\n";
  }
}

# Algorithms -- find_if()
print "Element 'second' was ",
  find_if($d->begin(), $d->end(), MyFind->new(what => 'second'))
    ? 'found' : 'not found', "\n";

# MyFind Unary Function -- used in find_if() above...
{
  package MyFind;
  use base qw(Class::STL::Utilities::FunctionObject::UnaryFunction);
  sub BEGIN { Class::STL::DataMembers->new( qw( what ) ); }
  sub new
  {
    my $self = shift;
    my $class = ref($self) || $self;
    $self = $class->SUPER::new(@_);
    bless($self, $class);
    $self->members_init(@_);
    return $self;
  }
  sub function_operator
  {
    my $self = shift;
    my $arg = shift;
    return $arg->data() eq $self->what() ? $arg : 0;
  }
}

# Algorithms -- count_if()
print "Number of elements matching /o/ = ",
  count_if($d->begin(), $d->end(), MyMatch->new(what => 'o')),
         "\n"; # prints '2' -- matches 'second' and 'fourth'

# Function Adaptors -- bind1st
remove_if($v->begin(), $v->end(), bind1st(equal_to(), $v->back()));
  # remove element equal to back() -- ie remove last element.
remove_if($v->begin(), $v->end(), MyMatch->new(what => '^fi'));
  # remove all elements that match reg-ex '^fi'

# Sort list according to elements cmp() function
$v->sort();

# Swap two elements
$v->swap($v->front(), $v->back());

# Queue containers -- FIFO
my $v = Class::STL::Containers::Queue->new();
```

```
$v->push($v->factory(data => ’first’));
$v->push($v->factory(data => ’second’));
$v->push($v->factory(data => ’third’));
$v->push($v->factory(data => ’fourth’));
$v->push($v->factory(data => ’fifth’));
print ’Back:’; MyPrint->new()->function_operator($v->back()); # Back:fifth
print ’Front:’; MyPrint->new()->function_operator($v->front()); # Front:first
$v->pop(); # pop element first in
$v->push($v->factory(data => ’sixth’));
print ’Back:’; MyPrint->new()->function_operator($v->back()); # Back:sixth
print ’Front:’; MyPrint->new()->function_operator($v->front()); # Front:second

# Iterators
for (my $i = $v->begin(); !$v->at_end(); $i++)
{
        MyPrint->new()->function_operator($i->p_element());
}

# Iterators -- reverse_iterator
my $ri = reverse_iterator($v->iter())->first();
while (!$ri->at_end())
{
        MyPrint->new()->function_operator($ri->p_element());
        $ri->next();
}

# MyMatch unary function -- used above in count_if()...
{
  package MyMatch;
  use base qw(Class::STL::Utilities::FunctionObject::UnaryFunction);
  sub BEGIN { Class::STL::DataMembers->new( qw( what ) ); }
  sub new
  {
    my $self = shift;
    my $class = ref($self) || $self;
    $self = $class->SUPER::new(@_);
    bless($self, $class);
    $self->members_init(@_);
    return $self;
  }
  sub function_operator
  {
    my $self = shift;
    my $arg = shift;
    return ($arg->data() =~ /@{[ $self->what() ]}/i) ? $arg : 0;
  }
}

# Vector container...
my $v = vector();
$v->push_back($v->factory(data => ’first’));
$v->push_back($v->factory(data => ’second’));
$v->push_back($v->factory(data => ’third’));
$v->push_back($v->factory(data => ’fourth’));
$v->push_back($v->factory(data => ’fifth’));

my $e = $v->at(0); # return pointer to first element.
print ’Element-0:’; MyPrint->new()->function_operator($e);# Element-0:first
$e = $v->at($v->size()-1); # return pointer to last element.
print ’Element-last:’; MyPrint->new()->function_operator($e);# Element-last:fifth
$e = $v->at(2); # return pointer to 3rd element (idx=2).
print ’Element-2:’; MyPrint->new()->function_operator($e);# Element-2:third

# Priority Queue
my $p = priority_queue();
$p->push($p->factory(priority => 10, data => ’ten’));
$p->push($p->factory(priority => 2, data => ’two’));
$p->push($p->factory(priority => 12, data => ’twelve’));
$p->push($p->factory(priority => 3, data => ’three’));
$p->push($p->factory(priority => 11, data => ’eleven’));
$p->push($p->factory(priority => 1, data => ’one’));
$p->push($p->factory(priority => 1, data => ’one-2’));
$p->push($p->factory(priority => 12, data => ’twelve-2’));
$p->push($p->factory(priority => 20, data => ’twenty’), $p->factory(priority => 0, data => ’zero’));
print "\$p->size()=", $p->size(), "\n";
print "\$p->top():"; MyPrint->new()->function_operator($p->top());
$p->top()->priority(7); # change priority for top element.
$p->refresh(); # refresh required after priority change.
$p->pop(); # remove element with highest priority.
print "\$p->top():"; MyPrint->new()->function_operator($p->top());
```

DESCRIPTION

These modules provide object container management with a framework similar to STL (Standard Template Library from C++). The usual container types are provided (list, vector, deque, queue, stack, priority_queue and also, tree) together with some basic algorithms (find_if, remove_if, foreach), utilities, and a very basic iterator type.

This package is usefull as a base framework for OO Perl applications development. It provides a number of shortcuts for building Classes and It will help you to get up and going very quickly with Perl OO program development.

CLASS Class::STL::Containers

Exports

vector, ***list***, ***deque***, ***queue***, ***priority_queue***, ***stack***, ***tree***.

CLASS Class::STL::Containers::Abstract

This is the *abstract* base class for all other container classes. Objects should not be constructed directly from this class, but from any of the derived container classes. Common functions are documented here.

Extends *Class::STL::Element*

new

container-ref new ([option-hash]);
container-ref new (container-ref);
container-ref new (element [, ...]);

The *new* function constructs an object for this class and returns a blessed reference to this object. All forms accept an optional *hash* containing any of the following key-value pairs: *name*, *element_type*.

The second form is a *copy constructor*. It requires another container reference as the argument and will return a copy of this container.

The third for requires one or more element refs as arguments. These elements will be copied into the newly constructed container.

factory

element-ref factory (%attributes);
The *factory* function constructs a new element object and returns a reference to this. The type of object created is as specified by the *element_type* container attribute. The *attributes* argument consists of a hash and is passed on to the element class *new* function. Override this function if you want to avoid the 'eval' call.

swap

void swap (element-1, element-2);
This function will swap the positions within the container of the two elements specified in the aruments.

erase

int erase (iterator-start [, iterator-finish]);
The *erase* function requires one starting iterator and an optional finish iterator as arguments. It will delete all the elements within the container within, and including, these two iterator positions. The *erase* funtion returns the number of elements deleted.

pop

void pop ( );
The *pop* function requires no arguments. It will remove the element at the *top* of the container.

push

void push (element [, ...]);
The *push* function requires one or more arguments consisting of elements. This will append the element(s) to the end of the container.

clear

void clear ( );
This function will delete all the elements from the container.

begin

iterator-ref begin ( );
The *begin* function constructs and returns a new iterator object which points to the first element within the container.

end

iterator-ref end ( );
The *end* function constructs and returns a new iterator object which points to the last element within the container. **Note that, unlike C++/STL, this object points to the last element and not *after the last element*.

rbegin

iterator-ref rbegin ( );
The *rbegin* function is the reverse of the *begin* function — the newly constructed iterator points to the last element.

rend

iterator-ref rend ( );
The *rend* function is the reverse of the *end* function — the newly constructed iterator points to the first element.

size

int size ( );
The *size* function requires no arguments. It will return an integer value containing the number of elements in the container.

empty

bool empty ( );
This function returns *'1'* if the container is empty (ie. contains no elements), and *'0'* if the container contains one or more elements.

to_array

array to_array ( );
The *to_array* function returns an array containing the elements (references) from the container.

eq

bool eq (container-ref);
The *eq* function compares the *elements* in this container with the *elements* in the container refered to by the argument *container-ref*. The elements are compared using the element *eq* function. The function will return *'1'* if both containers contain the same number of elements and all elements in one container are equal to, and in the same order as, all elements in the *container-ref* container.

ne

bool ne (container-ref);
Inverse of *eq* function.

gt

bool gt (container-ref);
Similar to *eq* function except comparison done for *greater-than* using elements *gt* function.

ge

bool ge (container-ref);
Similar to *eq* function except comparison done for *greater-than-or-equal* using elements *ge* function.

lt

bool lt (container-ref);
Similar to *eq* function except comparison done for *less-than* using elements *lt* function.

le

bool le (container-ref);
Similar to *eq* function except comparison done for *less-than-or-equal* using elements *le* function.

CLASS Class::STL::Containers::List

A list container can have elements pushed and popped from both ends, and also inserted at any location. Access to the elements is sequential.

Extends *Class::STL::Containers::Deque*

reverse

void reverse ( );
The *reverse* function will alter the order of the elements in list by reversing their order.

sort

void sort ( );
The *sort* function will alter the order of the elements in list by sorting the elements. Sorting is done based on the elements *cmp* comparison function.

Example

```
use Class::STL::Containers;

# Construct the list object:
my $list = list();

# Append elements to the list;
# Elements are constructed with the factory function:
$list->push_back($list->factory(data => ’first’));
$list->push_back($list->factory(data => ’second’));
$list->push_back($list->factory(data => ’third’));
$list->push_back($list->factory(data => ’fourth’));
$list->push_back($list->factory(data => ’fifth’));

# Display the number of elements in the list:
print "Size:", $list->size(), "\n"; # Size:5

# Reverse the order of elements in the list:
$list->reverse();

# Display the contents of the element at the front of the list:
print ’Front:’; MyPrint->new()->function_operator($list->back());

# Display the contents of the element at the back of the list:
print ’Back:’; MyPrint->new()->function_operator($list->back());

# Display the contents of all the elements in the list:
::foreach($list->begin(), $list->end(), MyPrint->new());

# Return an array of all elements-refs:
my @arr = $l1->to_array();

# Delete all elements from list:
$list->clear();

print "Size:", $list->size(), "\n"; # Size:0
print ’$list container is ’,
  $list->empty() ? ’empty’ : ’not empty’, "\n";

# MyPrint Unary Function -- used in ::foreach() above...
{
  package MyPrint;
  use base qw(Class::STL::Utilities::FunctionObject::UnaryFunction);
  sub function_operator
  {
    my $self = shift;
    my $arg = shift;
    print "Data:", $arg->data(), "\n";
  }
}
```

CLASS Class::STL::Containers::Vector

A vector allows for random access to its elements via the *at* function.

Extends *Class::STL::Containers::Abstract*

push_back

void push_back (element [, ...]);
The *push_back* function requires one or more arguments consisting of elements. This will append the element(s) to the end of the *vector*.

pop_back

void pop_back ( );
The *pop_back* function requires no arguments. It will remove the element at the *top* of the *vector*.

back

element-ref back ( );
The *back* function requires no arguments. It returns a reference to the element at the *back* of the *vector*.

front

The *front* function requires no arguments. It returns a reference to the element at the *front* of the *vector*.

at

element-ref at (index);
The *at* function requires an *index* argument. This function will return a reference to the element at the location within the *vector* specified by the argument *index*.

CLASS Class::STL::Containers::Deque

A double-ended container. Elements can be *pushed* and *popped* at both ends.

Extends *Class::STL::Containers::Vector*

push_front

void push_front (element [, ...]);
The *push_front* function requires one or more arguments consisting of elements. This will insert the element(s) to the front of the *deque*.

pop_front

void pop_front ( );
The *pop_front* function requires no arguments. It will remove the element at the *front* of the *deque*.

CLASS Class::STL::Containers::Queue

A queue is a FIFO (first-in-first-out) container. Elements can be *pushed* at the back and *popped* from the front.

Extends *Class::STL::Containers::Abstract*

push

void push (element [, ...]);
The *push* function requires one or more arguments consisting of elements. This will append the element(s) to the back of the *queue*.

pop

void pop ( );
The *pop* function requires no arguments. It will remove the element at the *front* of the *queue*. This is the earliest inserted element.

back

element-ref back ( );
The *back* function requires no arguments. It returns a reference to the element at the *back* of the *queue*. This is the element last inserted.

front

element-ref front ( );
The *front* function requires no arguments. It returns a reference to the element at the *front* of the *queue*. This is the earliest inserted element.

CLASS Class::STL::Containers::Stack

A stack is a LIFO (last-in-first-out) container. Elements can be *pushed* at the top and *popped* from the top.

Extends *Class::STL::Containers::Abstract*

push

void push (element [, ...]);
The *push* function requires one or more arguments consisting of elements. This will append the element(s) to the top of the *stack*.

pop

void pop ( );
The *pop* function requires no arguments. It will remove the element at the *top* of the *stack*. This is the last inserted element.

top

element-ref top ( );
The *top* function requires no arguments. It returns a reference to the element at the *top* of the *stack*. This is the last inserted element.

CLASS Class::STL::Containers::Tree

A tree is a hierarchical structure. Each element within a *tree* container can be either a simple element or another container object. The overridden *to_array* function will traverse the tree and return an array consisting of all the *nodes* in the tree.

Extends *Class::STL::Containers::Deque*

to_array

array to_array ( );

The overridden *to_array* function will traverse the tree and return an array consisting of all the element *nodes* in the tree container.

Examples

```
# Tree containers; construct two trees from
# previously construced containers:
my $t1 = tree($l1);
my $t2 = tree($l2);

# Construct a third tree:
my $tree = tree();

# Add other tree containers as elements to this tree:
$tree->push_back($tree->factory($t1));
$tree->push_back($tree->factory($t2));

# Search for element (’pink’) in tree:
if (my $f = find_if($tree->begin(), $tree->end(), MyFind->new(what => ’pink’))
  print "FOUND:", $f->data(), "\n";
} else {
  print "’pink’ NOT FOUND", "\n";
}

# Traverse tree returning all element nodes:
my @tarr = $tree->to_array();
```

CLASS Class::STL::Containers::PriorityQueue

A priority queue will maintain the order of the elements based on their priority, with highest priority elements at the top of the container. Elements contained in a priority queue must be of the type, or derived from, *Class::STL::Element::Priority*. This element type contains the attribute *priority*, and needs to have its value set whenever an object of this element type is constructed.

Extends *Class::STL::Containers::Vector*

Element Type *Class::STL::Element::Priority*

push

void push (element [, ...]);

The *push* function requires one or more arguments consisting of elements. This will place the element(s) in the queue according to their priority value.

pop

void pop_back ( );

The *pop* function requires no arguments. It will remove the element with the highest priority.

top

element-ref top ( );

The *top* function requires no arguments. It returns a reference to the element with the highest priority.

refresh

void refresh ( );

The *refresh* function should be called whenever the priority value for an element has been order. This will update the ordering of the elements if required.

CLASS Class::STL::Algorithms

This module contains various algorithm functions.

Exports

remove_if, ***find_if***, ***foreach***, ***transform***, ***count_if***.

The ***Algorithms*** package consists of various *static* algorithm functions.
The *unary-function-object* argument must be derived from *Class::STL::Utilities::FunctionObject::UnaryFunction*. Standard utility functions are provided in the *Class::STL::Utilities* module. A *unary-function-object* contains the function *function_operator*. This *function_operator* function will, in turn, be called by the algorithm for each element traversed. The algorithm will pass the element reference as the argument to the *function_operator* function.

remove_if

void remove_if (iterator-start, iterator-finish, unary-function-object);

The *remove_if* function will traverse the container starting from *iterator-start* and ending at *iterator-finish* and remove the elements that evaluate to true by the *unary-function-object*.

find_if

element-ref find_if (iterator-start, iterator-finish, unary-function-object);

The *find_if* function will traverse the container starting from *iterator-start* and ending at *iterator-finish* and return the first element that evaluate to true by the *unary-function-object*. If no elements evaluates to true then ‘o’ is returned.

foreach

void foreach (iterator-start, iterator-finish, unary-function-object);

The *foreach* function will traverse the container starting from *iterator-start* and ending at *iterator-finish* and execute the *unary-function-object* with the element passed in as the argument.

transform

void transform (iterator-start, iterator-finish, unary-function-object);

The *transform* function will traverse the container starting from *iterator-start* and ending at *iterator-finish* and execute the *unary-function-object* with the element passed in as the argument.

count_if

int count_if (iterator-start, iterator-finish, unary-function-object);

The *count_if* function will traverse the container starting from *iterator-start* and ending at *iterator-finish* and return a count of the elements that evaluate to true by the *unary-function-object*.

Examples

```
use Class::STL::Containers;
use Class::STL::Algorithms;
use Class::STL::Utilities;

# Display all elements in list container ’$list’
# using unary-function-object ’MyPrint’ and algorithm ’foreach’:
::foreach($list->begin(), $list->end(), MyPrint->new());

# Algorithms -- remove_if()
# Remove element equal to back() -- ie remove last element:
remove_if($list->begin(), $list->end(), bind1st(equal_to(), $list->back()));

# Remove all elements that match regular expression ’^fi’:
remove_if($v->begin(), $v->end(), MyMatch->new(what => ’^fi’));

# Search for element (’pink’) in tree:
if (my $f = $tree->find_if(MyFind->new("pink"))) {
  print "FOUND:", $f->data(), "\n";
} else {
  print "’pink’ NOT FOUND", "\n";
}

# MyPrint unary function object:
{
  package MyPrint;
  use base qw(Class::STL::Utilities::FunctionObject::UnaryFunction);
  sub function_operator
  {
    my $self = shift;
    my $arg = shift;
    print "Data:", $arg->data(), "\n";
  }
}
# MyFind Unary function object:
{
  package MyFind;
  use base qw(Class::STL::Utilities::FunctionObject::UnaryFunction);
  sub BEGIN { Class::STL::DataMembers->new( qw( what ) ); }
  sub new
  {
    my $self = shift;
    my $class = ref($self) || $self;
    $self = $class->SUPER::new(@_);
    bless($self, $class);
    $self->members_init(@_);
    return $self;
  }
  sub function_operator
  {
    my $self = shift;
    my $arg = shift;
    return $arg->data() eq $self->what() ? $arg : 0;
  }
}
{
  package MyMatch;
  use base qw(Class::STL::Utilities::FunctionObject::UnaryFunction);
  sub BEGIN { Class::STL::DataMembers->new( qw( what ) ); }
  sub new
  {
    my $self = shift;
    my $class = ref($self) || $self;
    $self = $class->SUPER::new(@_);
    bless($self, $class);
    $self->members_init(@_);
    return $self;
  }
  sub function_operator
  {
    my $self = shift;
    my $arg = shift;
    return ($arg->data() =~ /@{[ $self->what() ]}/i) ? $arg : 0;
  }
}
```

CLASS Class::STL::Utilities

Exports

equal_to, ***not_equal_to***, ***greater***, ***greater_equal***, ***less***, ***less_equal***, ***compare***, ***bind1st***, ***bind2nd***, ***mem_fun***.

This module contains various utility function objects. Each object will be constructed automatically when the function name (eg. ‘equal_to’) is used. Each of the function objects are derived from either *Class::STL::Utilities::UnaryFunction* or *Class::STL::Utilities::BinaryFunction*. These classes contain the function *do* which requires one argument consisting of an element reference. Any value (including *void*) can be returned. The *unary* objects contain the attribute *arg*, and the *binary* objects contain the attributes *arg1* and *arg2*. These attributes are initialised when the function object is constructed and are available to the function object.

equal_to

This function-object will return the result of *equality* between its argument and the object *arg* attribute‘s value. The element‘s *eq* function is used for the comparison.

not_equal_to

This function is the inverse of *equal_to*.

greater

This function-object will return the result of *greater-than* comparison between its argument and the object *arg* attribute‘s value. The element‘s *gt* function is used for the comparison.

greater_equal

This function-object will return the result of *greater-than-or-equal* comparison between its argument and the object *arg* attribute‘s value. The element‘s *ge* function is used for the comparison.

less

This function-object will return the result of *less-than* comparison between its argument and the object *arg* attribute‘s value. The element‘s *lt* function is used for the comparison.

less_equal

This function-object will return the result of *less-than-or-equal* comparison between its argument and the object *arg* attribute‘s value. The element‘s *le* function is used for the comparison.

compare

This function-object will return the result of *compare* comparison between its argument and the object *arg* attribute‘s value. The element‘s *cmp* function is used for the comparison.

matches

This function-object will return the result of regular expression comparison between its argument and the object *arg* attribute‘s (regular expression) value. The element‘s *match* function is used for the comparison.

bind1st

bind2nd

mem_fun

CLASS Class::STL::Iterators

This module contains the iterator classes.

Exports

iteratror, ***reverse_iteratror***, ***forward_iteratror***.

new

first

next

last

prev

set

jump

at_end

eq

ne

lt

le

gt

ge

cmp

Examples

```
# Using overoaded inrement operator:
for (my $i = $p->begin(); !$i->at_end(); $i++)
{
         MyPrint->new()->function_operator($i->p_element());
}

# Using overoaded decrement operator:
for (my $i = $p->end(); !$i->at_end(); --$i)
{
         MyPrint->new()->function_operator($i->p_element());
}

# Reverse iterator:
my $ri = reverse_iterator($p->iter())->first();
while (!$ri->at_end())
{
  MyPrint->new()->function_operator($ri->p_element());
  $ri->next();
}
```

SEE ALSO

This framwork mimicks the C++/STL Container-Iterators-Algorithms library.

AUTHOR

m gaffiero, <gaffie@users.sourceforge.net<gt>

COPYRIGHT AND LICENSE