Programmer’s Reference

version 0.34

gaffie@users.sourceforge.net

Class::STL::Containers

Table of Contents
Class::STL::Containers

NAME

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

SYNOPSIS

```
use stl;

# Deque container...
my $d = stl::deque(qw(first second third fourth));
$d->push_back($d->factory('fifth'));
$d->push_front($d->factory('seventh'));
$d->pop_front(); # remove element at front.
$d->pop_back(); # remove element at back.
stl::for_each($d->begin(), $d->end(), ptr_fun('::myprint'));

sub myprint { print "Data:", @_, "\n"; }

# Copy constructor...
my $d_copy = stl::deque($d);

# Algorithms -- find_if()
print "Element 'second' was ",
  stl::find_if($d->begin(), $d->end(), stl::bind1st(stl::equal_to(), 'second'))
    ? 'found' : 'not found', "\n";

# Algorithms -- count_if()
print "Number of elements matching /o/ = ",
  stl::count_if($d->begin(), $d->end(), stl::bind2nd(stl::matches(), 'o')),
         "\n"; # prints '2' -- matches 'second' and 'fourth'

# Algorithms -- transform()
stl::transform($d->begin(), $d->end(), $d2->begin(), stl::ptr_fun('ucfirst'));
stl::transform($d->begin(), $d->end(), $d2->begin(), $d3->begin(), stl::ptr_fun_binary('::mybfun'));
sub mybfun { return $_[0] . '-' . $_[1]; }

# Function Adaptors -- bind1st
stl::remove_if($v->begin(), $v->end(), stl::bind1st(stl::equal_to(), $v->back()));
  # remove element equal to back() -- ie remove last element.
stl::remove_if($v->begin(), $v->end(), stl::bind2nd(stl::matches(), '^fi'));
  # remove all elements that match reg-ex '^fi'

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

# Queue containers -- FIFO
my $v = stl::queue(qw(first second third fourth fifth));
print 'Back:', $v->back()->data(), "\n" # Back:fifth
print 'Front:', $v->front()->data(), "\n" # Front:first
$v->pop(); # pop element first in
$v->push($v->factory('sixth')), "\n"
print 'Back:', $v->back()->data(), "\n" # Back:sixth
print 'Front:', $v->front()->data(), "\n" # Front:second

# Iterators
for (my $i = $v->begin(); !$i->at_end(); ++$i)
{
         print "Data:", $i->p_element()->data();
}

# Iterators -- reverse_iterator
my $ri = stl::reverse_iterator($v->iter())->first();
while (!$ri->at_end())
{
         print "Data:", $ri->p_element()->data();
         ++$ri;
}

# Inserters
my $three2one = stl::list(qw(3 2 1));
my $four2six = stl::list(qw(4 5 6));
my $seven2nine = stl::list(qw(7 8 9));
my $result = stl::list();
stl::copy($three2one->begin(), $three2one->end(), stl::front_inserter($result));
stl::copy($seven2nine->begin(), $seven2nine->end(), stl::back_inserter($result));
my $iseven = stl::find($result->begin(), $result->end(), 7);
stl::copy($four2six->begin(), $four2six->end(), stl::inserter($result, $iseven));
# $result now contains (1, 2, 3, 4, 5, 6, 7, 8, 9);

# Vector container...
my $v = stl::vector(qw(first second third fourth fifth));
```

```
my $e = $v->at(0); # return pointer to first element.
print 'Element-0:', $e->data(), "\n";   # Element-0:first
$e = $v->at($v->size()-1); # return pointer to last element.
print 'Element-last:', $e->data(), "\n";  # Element-last:fifth
$e = $v->at(2); # return pointer to 3rd element (idx=2).
print 'Element-2:', $e->data(), "\n";   # Element-2:third

# Priority Queue
my $p = stl::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():", $p->top(), "\n";
$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():", $p->top(), "\n";

# Clone $d container into $d1...
my $d1 = $d->clone();

my $d2 = stl::deque(qw(sixth seventh eight));

# Append $d container to end of $d2 container...
$d2 += $d;

# DataMembers -- Class builder helper...
{
  package MyClass;
  use Class::STL::ClassMembers (
      qw(attrib1 attrib2), # data members
      Class::STL::ClassMembers::DataMember->new(
          name => 'attrib3', default => '100', validate => '^\d+$'), # data member with attributes
      Class::STL::ClassMembers::DataMember->new(
          name => 'attrib4', default => 'med', validate => '^(high|med|low)$'),
  );
  use Class::STL::ClassMembers::Constructor;  # produce class new() function
}
my $cl = MyClass->new(attrib1 => 'hello', attrib2 => 'world');
print $cl->attrib1(), " ", $cl->attrib2(), "\n"; # 'hello world'
$cl->attrib1(ucfirst($cl->attrib1));
$cl->attrib2(ucfirst($cl->attrib2));
print $cl->attrib1(), " ", $cl->attrib2(), "\n"; # 'Hello World'
$cl->attrib4('avg'); # Causes progam to die with '** Function attrib2 value failed validation...'
```

DESCRIPTION

This package provides a framework for rapid Object Oriented Perl application development. It consists of a number of base classes that are similar to the C++/STL framework, plus a number of *helper* classes which provide the *glue* to transparently generate common functions, and will enable you to put your Perl application together very quickly.

The *STL* functionality provided consists of ***containers***, ***algorithms***, ***utilities*** and ***iterators*** as follows:

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

Iterators
iterator, bidirectional_iterator, reverse_iterator, forward_iterator.

Algorithms
find, find_if, for_each, transform, count, count_if, copy, copy_backward, remove, remove_if, remove_copy, remove_copy_if, replace, replace_if, replace_copy, replace_copy_if.

Utilities
equal_to, not_equal_to, greater, greater_equal, less, less_equal, compare, bind1st, bind2nd, mem_fun, ptr_fun, ptr_fun_binary, matches, matches_ic, logical_and, logical_or, multiplies, divides, plus, minus, modulus.

Differences From C++/STL

Most of the functions have the same arguments and return types as their STL equivalent. There are some differences though between the C++/STL and this implementation:

Iterators and the *end()* function
An *iterator* object points to a numeric position within the container, and not to an *element*. If new elements are inserted to, or removed from, a postion preceding the iterator, then the iterator will point to the same *position* but to a different element.

The *end* function will return a newly constructed iterator object which will point to the *last* element within the container, unlike the C++/STL equivalent which points to *after* the last element.

The *tree* Container
This container provides a hierarchical tree structure. Each element within a *tree* container can be either a simple element or another container object. The *algorithms* and overridden *to_array* functions will traverse the tree and pocess all element *nodes* within the tree.

Utilities *matches*, *matches_ic* functions
These utilities provide unary functions for regular expression matching. The first or second argument will be a regular expression string. The *match_ic* provides case insensitive matching.

Container *append* function
This function and the overridden +, += operators will combine the two containers together.

The *clone* function
This function returns a newly constructed object that is a copy of its caller object.

The Container *to_array* function
This function will return an array consisting of all element objects within the container.

Container *element* type
All containers contain collections of objects which are of type ***Class::STL::Element***, or classes derived from this type. The container classes are themselves, ultimately, derived from this *element* type.

CLASS Class::STL::ClassMembers

These *helper* classes can be used to generate code for various basic class functions. This module requires an import list consisting of target data member names or
Class::STL::ClassMembers::DataMember objects. When using *ClassMembers* **ALL** data members should be included in order for the generated *clone* and *swap* functions to function correctly. The constructor function code can be produces as well by using the package
Class::STL::ClassMembers::Constructor, or ***Class::STL::ClassMembers::SingletonConstructor*** to create a singleton class type.

The following target member functions will be generated and made available to the class:

Data Member Accessor Get/Put Function
This function will have the same name as the data member and should be used to *set* or *get* the value for the data member. Pass the value as the argument when setting the value for a data member. For *comlpex* data members with a ***validate*** attribute, a validation check will be performed when attempting to set the member value by matching the value against the *validate* regular expression string.

Class *members_init()* Function
This function should be called in the target class's *new* function after *$self* has been blessed. It will perform the necessary data members initialisation.

Class *clone()* Function

This function will construct and return an object containing a copy of the caller object.

Class *swap()* Function
This function requires one argument consisting of an object of the same type as the caller. It will swap the caller object with this *other* object.

Class *members()* Function
This function will return a pointer to an anonymous hash containing the data member names (as the key) and data member attibutes array list consisting of ***default*** and ***validate*** attribute fields in that order. All data members, including inherited members are contained in this hash.

Class *members_local()* Function
Same as ***members*** function except that only the data members local to the class are contained in the hash returned.

Class::STL::ClassMembers::DataMember
For more complex data members, this class may be used to provide additional information about the member. This information consist of: ***name***, ***default***, and ***validate***. The ***name*** attribute contains the member name; the ***default*** attribute contains a default value for the member when initialised; the ***validate*** attribute consists of a regular expression string that will be used to validate the member value by matching it to this regex string.

Class::STL::ClassMembers::Constructor
The constructor function with the name ***new()*** will be produced for a package that uses this module. It is recomended that this constructor is produced for any class (package) that uses the ***ClassMembers*** package to produce the data members. This will ensure that the correct calls are done during construction and copy-construction of an object. This constructor will make a call to the `static` user member function ***new_extra*** if it exists in the calling class. The ***new_extra*** function will have the object reference passed as the first argument.

Class::STL::ClassMembers::SingletonConstructor
Use this package to produce a singleton class. This constructor will ensure that only one instance of this class will be constructed.

Example

```
{
  package MyClass;
  use Class::STL::ClassMembers (
      qw(attrib1 attrib2),
      Class::STL::ClassMembers::DataMember->new(
          name => 'attrib3', default => '100', validate => '^\d+$'),
      Class::STL::ClassMembers::DataMember->new(
          name => 'attrib4', default => 'med', validate => '^(high|med|low)$'),
  );
  use Class::STL::ClassMembers::Constructor;  # produce class new() function
}
my $cl = MyClass->new(attrib1 => 'hello', attrib2 => 'world');
print $cl->attrib1(), " ", $cl->attrib2(), "\n"; # 'hello world'
$cl->attrib1(ucfirst($cl->attrib1));
$cl->attrib2(ucfirst($cl->attrib2));
print $cl->attrib1(), " ", $cl->attrib2(), "\n"; # 'Hello World'
$cl->attrib4('avg'); # Causes progam to die with '** Function attrib2 value failed validation...'
```

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 ([named-argument-list]);
container-ref new (container-ref);
container-ref new (element [, ...]);
container-ref new (iterator-start [, iterator-finish]);
container-ref new (raw-data [, ...]);

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 named arguments: *element_type*. The *element_type* defines the class type of element objects that the container will hold. *element_type* will default to ***Class::STL::Element*** if not specified; when specified, the type must be derived from ***Class::STL::Element***.

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

The fourth form requires one *start* iterator and an optional *finish* iterator. All the element objects with, and including, the *start* and *finish* (or *end* if not specified) positions will be copied into the newly constructed container.

The fifth form accepts a list of raw data values. Each of these values will be stored inside a ***Class::STL::Element*** object constructed by the container's *factory* function, with the element's *data* member containing the raw data value.

clone

Returns a newly constructed object which is identical to the calling (this) object.

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.

erase

iterator 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 in the container within, and including, these two iterator positions. The *erase* funtion returns and iterator pointing to the element following the last deleted element.

insert

void insert (position, iterator-start, iterator-finish);
void insert (position, iterator-start);
void insert (position, element [, ...]);
void insert (position, size, element);
The first form will insert copies of elements within the *iterator-start* and *iterator-finish* positions before *position*.

The second form will insert copies of elements within the *iterator-start* and *end* positions before *position*

The third form will insert the element, or elements (*not copies*) before *position*.

The fourth form will insert *size* copies of *element* before *position*.

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 *front* element within the container.

end

iterator-ref end ( );
The *end* function constructs and returns a new iterator object which points to the ***back*** 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.

operator +, ***operator +=***

Append containers.

operator ==

Containers equality comparison.

operator !=

Containers non-equality comparison.

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 stl;

# Construct the list object:
my $list = list(qw( first second third fourth 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:', $list->front(), "\n";

# Display the contents of the element at the back of the list:
print 'Back:', $list->back(), "\n";

# Display the contents of all the elements in the list:
for_each($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 for_each() 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(), bind1st(equal_to(), '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***, ***for_each***, ***transform***, ***count_if***, ***find***, ***count***, ***copy***, ***copy_backward***, ***remove***, ***remove_copy***, ***remove_copy_if***, ***replace***, ***replace_if***, ***replace_copy***, ***replace_copy_if***, ***generate***, ***generate_n***, ***fill***, ***fill_n***, ***equal***, ***reverse***, ***reverse_copy***, ***rotate***, ***rotate_copy***, ***partition***, ***stable_partition***, ***min_element***, ***max_element***, ***unique***, ***unique_copy***, ***adjacent_find***

The ***Algorithms*** package consists of various *static* algorithm functions.
The *unary-function* / *binary-function* argument must be derived from *Class::STL::Utilities::FunctionObject::UnaryFunction* and *Class::STL::Utilities::FunctionObject::BinaryFunction* respectively. Standard utility functions are provided in the *Class::STL::Utilities* module. A 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.

for_each

void for_each (iterator-start, iterator-finish, unary-function);

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

transform

void transform (iterator-start, iterator-finish, iterator-result, unary-function);
void transform (iterator-start, iterator-finish, iterator-start2, iterator-result, binary-function);

The *transform* functions has two forms. The first form will traverse the container starting from *iterator-start* and ending at *iterator-finish* and execute the *unary-function* with the element passed in as the argument, producing *iterator-result*.

The second form will traverse two containers with the second one starting from *iterator-start2*. The *binary-function* will be called for each pair of elements. The resulting elements will be placed in *iterator-result*.

count

int count (iterator-start, iterator-finish, element-ref);

count_if

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

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*.

find

iterator-ref find (iterator-start, iterator-finish, element-ref);

find_if

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

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

copy

void copy (iterator-start, iterator-finish, iterator-result);

copy_backward

void copy_backward (iterator-start, iterator-finish, iterator-result);

remove

void remove (iterator-start, iterator-finish, element-ref);

remove_if

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

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*.

remove_copy
void remove_copy (iterator-start, iterator-finish, iterator-result, element-ref);

remove_copy_if
void remove_copy_if (iterator-start, iterator-finish, iterator-result, unary-function);

replace
void replace (iterator-start, iterator-finish, old-element-ref, new-element-ref);

replace_if
void replace_if (iterator-start, iterator-finish, unary-function, new-element-ref);

replace_copy
void replace_copy (iterator-start, iterator-finish, iterator-result, old-element-ref, new-element-ref);

replace_copy_if
void replace_copy_if (iterator-start, iterator-finish, iterator-result, unary-function, new-element-ref);

generate
void generate (iterator-start, iterator-finish, generator-function);

generate_n
void generate_n (iterator-start, size, generator-function);

fill
void fill (iterator-start, iterator-finish, element-ref);

fill_n
void fill_n (iterator-start, size, element-ref);

equal
bool equal (iterator-start, iterator-finish, iterator-result [, binary-function]);

reverse
void reverse (iterator-start, iterator-finish);

reverse_copy
void reverse_copy (iterator-start, iterator-finish, iterator-result);

rotate
void rotate (iterator-start, iterator-mid, iterator-finish);

rotate_copy
void rotate_copy (iterator-start, iterator-mid, iterator-finish, iterator-result);

partition
void partition (iterator-start, iterator-finish, [, unary-predicate]);

stable_partition
void stable_partition (iterator-start, iterator-finish, [, unary-predicate]);

min_element

iterator min_element (iterator-start, iterator-mid [, binary-function]);

max_element
iterator max_element (iterator-start, iterator-mid [, binary-function]);

unique
iterator unique (iterator-start, iterator-finish, [, binary-function]);

unique_copy
iterator unique_copy (iterator-start, iterator-finish, iterator-result [, binary-function]);

adjacent_find
iterator ajacent_find (iterator-start, iterator-finish, [, binary-predicate]);

Examples

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

# Display all elements in list container '$list'
# using unary-function 'myprint' and algorithm 'for_each':
for_each($list->begin(), $list->end(), ptr_fun('::myprint'));
sub myprint { print "Data:", @_, "\n"; }

# 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(), bind2nd(matches(), '^fi'));

# Search for element ('pink') in tree:
if (my $f = $tree->find_if($tree->begin(), $tree->end(), bind1st(equal_to(), "pink"))) {
  print "FOUND:", $f->p_element()->data(), "\n";
} else {
  print "'pink' NOT FOUND", "\n";
}
```

CLASS Class::STL::Utilities

Exports
equal_to, ***not_equal_to***, ***greater***, ***greater_equal***, ***less***, ***less_equal***, ***compare***, ***bind1st***, ***bind2nd***, ***mem_fun***, ***ptr_fun***, ***ptr_fun_binary***, ***matches***, ***matches_ic***, ***logical_and***, ***logical_or***, ***multiplies***, ***divides***, ***plus***, ***minus***, ***modulus***.

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::FunctionObject::UnaryFunction* or *Class::STL::Utilities::FunctionObject::BinaryFunction*.

equal_to
Binary predicate. 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
Binary predicate. This function is the inverse of *equal_to*.

greater
Binary predicate. 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

Binary predicate. 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

Binary predicate. 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

Binary predicate. 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

Binary predicate. 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

Binary predicate. This function-object will return the result (true or false) of the regular expression comparison between its first argument and its second argument which contains a regular expression string.

matches_ic

Binary predicate. Case-insensitive version of the *matches* function.

bind1st

Unary function. This function requires two arguments consisting of a *binary-function-object* and a element or value argument. It will produce a *unary-function* object whose *function_operator* member will call the *binary-function* with *argument* as the first argument.

bind2nd

Unary function. This function requires two arguments consisting of a *binary-function-object* and a element or value argument. It will produce a *unary-function* object whose *function_operator* member will call the *binary-function* with *argument* as the second argument.

mem_fun

This function requires one argument consisting of the class member function name (string). It will construct an object whose *function_operator* member will require an element object to be passed as the first argument. It will call the elements's member function as specified by the *mem_fun* argument.

ptr_fun

Unary function. This function requires one argument consisting of a global function name (string).

ptr_fun_binary

Binary function. This function requires one argument consisting of global function name (string).

logical_and

Binary predicate.

logical_or

Binary predicate.

multiplies

Binary function. This function-object will return the result of *multiply* between its two element arguments. The element's *mult* function is used for the calculation. It will return a newly construced element object containing the result.

divides

Binary function. This function-object will return the result of *division* between its two element arguments. The element's *div* function is used for the calculation. It will return a newly construced element object containing the result.

plus

Binary function. This function-object will return the result of *plus* between its two element arguments. The element's *add* function is used for the calculation. It will return a newly construced element object containing the result.

minus

Binary function. This function-object will return the result of *subtract* between its two element arguments. The element's *subtract* function is used for the calculation. It will return a newly construced element object containing the result.

modulus.

Binary function. This function-object will return the result of *modulus* between its two element arguments. The element's *mod* function is used for the calculation. It will return a newly construced element object containing the result.

CLASS Class::STL::Iterators

This module contains the iterator classes.

Exports

iterator, ***bidirectional_iterator***, ***reverse_iterator***, ***forward_iterator***, **++**, **—**, **==**, ***!=***, **>=**, **<=**, **+**, **+=**, **-**, **-=**, ***distance***, ***advance***, ***front_inserter***, ***back_inserter***, ***inserter***.

new

p_container

Returns a reference to the container that is associated with the iterator.

p_element

This function will return a reference to the element pointed to by the iterator.

distance

Static function. This function will return the *distance* between two iterators. Both iterators must be from the same container. *Iterator-finish* must be positioned after *iterator-first*.
int distance (iterator-start, iterator-finish]);

advance

Static function. Moves the iterator forward, or backwards if size is negative.

iterator advance (iterator, size);

inserter

Static function.

iterator inserter (container, iterator);

front_inserter

Static function.

iterator front_inserter (container);

back_inserter

Static function.

iterator back_inserter (container);

first
next
last
prev
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

Sourceforge Project Page: `http://sourceforge.net/projects/pstl`

AUTHOR

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

COPYRIGHT AND LICENSE