



# Programmer's Reference

by *M Gaffiero*

---

[gaffie@users.sourceforge.net](mailto:gaffie@users.sourceforge.net)

Class::STL::Containers

0.11



# Table of Contents

## Class::STL::Containers

NAME	1
SYNOPSIS	1
DESCRIPTION	3
Containers	3
Iterators	3
Algorithms	3
Utilities	3
Differences From C++/STL	3
The end() Function	3
The tree Container	3
Utilities matches, matches_ic functions	3
Container append function	3
The clone function	3
The Container to_array function	3
Container element type	3
CLASS Class::STL::DataMembers	3
Data Member Accessor Get/Put Function	4
Class members_init() Function	4
Class clone() Function	4
Class swap() Function	4
Class::STL::DataMembers::Attributes	4
Example	4
CLASS Class::STL::Containers	4
Exports	4
CLASS Class::STL::Containers::Abstract	5
Extends Class::STL::Element	5
new	5
clone	5
factory	5
erase	5
insert	5
pop	6
push	6
clear	6
begin	6
end	6
rbegin	6
rend	6
size	6
empty	6
to_array	6
eq	6
ne	6
operator +, operator +=	7
operator ==	7
operator !=	7
CLASS Class::STL::Containers::List	7
Extends Class::STL::Containers::Deque	7
reverse	7
sort	7
Example	7
CLASS Class::STL::Containers::Vector	8
Extends Class::STL::Containers::Abstract	8

push_back	8
pop_back	8
back	8
front	8
at	8
CLASS Class::STL::Containers::Deque	8
Extends Class::STL::Containers::Vector	8
push_front	8
pop_front	8
CLASS Class::STL::Containers::Queue	8
Extends Class::STL::Containers::Abstract	8
push	8
pop	9
back	9
front	9
CLASS Class::STL::Containers::Stack	9
Extends Class::STL::Containers::Abstract	9
push	9
pop	9
top	9
CLASS Class::STL::Containers::Tree	9
Extends Class::STL::Containers::Deque	9
to_array	9
Examples	10
CLASS Class::STL::Containers::PriorityQueue	10
Extends Class::STL::Containers::Vector	10
Element Type Class::STL::Element::Priority	10
push	10
pop	10
top	10
refresh	10
CLASS Class::STL::Algorithms	10
Exports	10
for_each	11
transform	11
count	11
count_if	11
find	11
find_if	11
copy	11
copy_backward	11
remove	11
remove_if	11
remove_copy	11
remove_copy_if	12
replace	12
replace_if	12
replace_copy	12
replace_copy_if	12
Examples	12
CLASS Class::STL::Utilities	12
Exports	12
equal_to	12
not_equal_to	12
greater	12
greater_equal	13
less	13
less_equal	13
compare	13

matches	13
matches_ic	13
bind1st	13
bind2nd	13
mem_fun	13
ptr_fun	13
ptr_fun_binary	13
logical_and	13
logical_or	13
multiplies	14
divides	14
plus	14
minus	14
modulus.	14
CLASS Class::STL::Iterators	14
Exports	14
new	14
first	14
next	14
last	14
prev	14
set	14
jump	14
at_end	14
eq	14
ne	14
lt	14
le	14
gt	14
ge	14
cmp	14
Examples	14
SEE ALSO	15
AUTHOR	15
COPYRIGHT AND LICENSE	15



**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;
use Class::STL::DataMembers;

# Deque container...
my $d = 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.
$d->pop_back(); # remove element at back.
for_each($d->begin(), $d->end(), ptr_fun('::myprint'));

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

# Copy constructor...
my $d_copy($d);

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

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

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

# 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(), bind2nd(matches(), '^fi'));
# remove all elements that match reg-ex '^fi'

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

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

# Vector container...
my $v = 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 = 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 = deque(qw(sixth seventh eight));

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

# DataMembers -- Class builder helper...
{
    package MyClass;
    sub BEGIN
    {
        Class::STL::DataMembers->new(qw( attrib1 attrib2 ), # simple data members
        Class::STL::DataMembers::Attributes->new( # complex data members
            name => 'attrib3', default => '100', validate => '^d+$',
            name => 'attrib4', default => 'med', validate => '^(high|med|low)$'
        )); # our named arguments
    }
    sub new
    {
        my $self = shift;
        my $class = ref($self) || $self;
        $self = $class->SUPER::new(@_);
        bless($self, $class);
        $self->members_init(@_); # initialise any named arguments.
        return $self;
    }
}
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 program to die with '** Function attrib2 value failed validation...'

# MakeFind -- Class builder helper...
# MakeFind is only available to classes derived from Class::STL::Containers::Abstract.
{
    package MyContainerClass;
    use base qw(Class::STL::Containers::Vector);
    sub begin
    {
        Class::STL::DataMembers->new(qw( name value ));
        Class::STL::Containers::MakeFind(qw( name )); # Create find() function based on 'name' members.
    }
    sub new
    {
        my $self = shift;
        my $class = ref($self) || $self;
        $self = $class->SUPER::new(@_);
        bless($self, $class);
        $self->members_init(@_); # initialise any named arguments.
        return $self;
    }
}
my $cl = MyContainerClass->new();
$cl->push_back($cl->factory(name => 'red', value => 100));
$cl->push_back($cl->factory(name => 'blue', value => 102));
$cl->push_back($cl->factory(name => 'green', value => 104));
$cl->push_back($cl->factory(name => 'orange', value => 107));
print "Green value=", $cl->find('green')->value(), "\n"; # 'Green value=104'
# find element where name eq 'green';

```



**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:

The *end()* Function

This 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 process 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::DataMembers**

This *helper* class can be used to generate code for various basic class functions. An object of this class should be constructed (and destroyed) in the **BEGIN** subroutine for the target class. Construction is done

via its *new* function. This function requires a list consisting of target data member names, or *Class::STL::DataMembers::Attributes* objects. When using *DataMembers* **ALL** data members should be included in order for the generated *clone* and *swap* functions to function correctly.

A debug version of *new*, *new\_debug* can be used to display the generated code. 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 *complex* 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::STL::DataMembers::Attributes

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.

## Example

```
use Class::STL::DataMembers;
{
    package MyClass;
    sub BEGIN
    {
        Class::STL::DataMembers->new(qw( attrib1 attrib2 ), # simple data members
        Class::STL::DataMembers::Attributes->new( # complex data members
            name => 'attrib3', default => '100', validate => '^\d+$',
            name => 'attrib4', default => 'med', validate => '^(high|med|low)$'
        )); # our named arguments
    }
    sub new
    {
        my $self = shift;
        my $class = ref($self) || $self;
        $self = $class->SUPER::new(@_);
        bless($self, $class);
        $self->members_init(@_); # initialise any named arguments.
        return $self;
    }
}
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 program 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: *data\_type*. The *data\_type* defines the class type of element objects that the container will hold.

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**

```
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* function returns the number of elements deleted.

**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 referred 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 Class::STL::Containers;

# 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 = $list->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 constructed 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,*

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.

**for\_each**

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

The *for\_each* 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, 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-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*.

**find**

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

**find\_if**

```
iterator-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 an *iterator* pointing to the first element that evaluate to true by the *unary-function-object*. 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-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*.

**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-object );
```

**replace**

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

**replace\_if**

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

**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:", $_[0], "\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 constructed 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 constructed 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 constructed 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 constructed 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 constructed element object containing the result.

**CLASS Class::STL::Iterators**

This module contains the iterator classes.

**Exports**

*iterator*, *bidirectional\_iterator*, *reverse\_iterator*, *forward\_iterator*, ++, —, ==, !=, >=, <=.

*new*

*first*

*next*

*last*

*prev*

*set*

*jump*

*at\_end*

*eq*

*ne*

*lt*

*le*

*gt*

*ge*

*cmp*

**Examples**

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

# Using overloaded 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

Copyright (C) 2006 by Mario Gaffiero

This file is part of Class::STL::Containers(TM).

Class::STL::Containers is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

Class::STL::Containers is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with Class::STL::Containers; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

