

Vector framework: Electronic appendix

Jyrki Katajainen and Bo Simonsen

*Department of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen East, Denmark*

Abstract. This is the electronic appendix to our article “Adaptable component frameworks: Using `vector` from the C++ standard library as an example”. The report contains the programs related to our `vector` framework.

Keywords. CPH STL, vector

Table of contents

1	Vector	1
1.1	Vector/Code/stl-vector.h++	1
1.2	Vector/Code/stl-vector.i++	3
2	Framework	10
2.1	Vector-framework/Code/vector-framework.h++	10
2.2	Vector-framework/Code/vector-framework.i++	12
3	Encapsulators	17
3.1	Vector-framework/Code/direct-encapsulator.h++	17
3.2	Vector-framework/Code/indirect-encapsulator.h++	17
3.3	Vector-framework/Code/doubly-indirect-encapsulator.h++	18
4	Encapsulator factories	20
4.1	Vector-framework/Code/encapsulator-factory.h++	20
5	General surrogate	22
5.1	Vector-framework/Code/general-surrogate.h++	22
6	Other proxies	22
6.1	Vector-framework/Code/reference-proxy.h++	22
6.2	Proxy/Code/allocator-proxy.h++	25
7	Iterators	27
7.1	Iterator/Code/rank-iterator.h++	27
7.2	Iterator/Code/rank-iterator.i++	29
7.3	Iterator/Code/proxy-iterator.h++	32
7.4	Iterator/Code/proxy-iterator.i++	34
8	Kernels	39
8.1	Vector-framework/Code/dynamic-array.h++	39
8.2	Vector-framework/Code/hashed-array-tree.h++	41
8.3	Vector-framework/Code/levelwise-allocated-pile.h++	44
9	Copying	46
9.1	Vector-framework/Code/slot-swap.i++	46
9.2	Algorithm/Code/uninitialized-copy.i++	47
10	Tests	48
10.1	Vector-framework/Test/smoke-test.c++	48
11	UNIX makefile	51
11.1	Vector-framework/Test/makefile	51
12	Benchmarks	53
12.1	Vector-framework/Benchmark/drive.c++	53
12.2	Vector-framework/Benchmark/experiment.c++	55
12.3	Vector-framework/Benchmark/plot-push-back.py	57
12.4	Vector-framework/Benchmark/plot-pop-back.py	60
12.5	Vector-framework/Benchmark/plot-sequential-access.py	62

12.6 Vector-framework/Benchmark/plot-random-access.py ...	64
12.7 Vector-framework/Benchmark/plot-insert.py.....	67
12.8 Vector-framework/Benchmark/makefile.....	69

1. Vector

1.1 Vector/Code/stl-vector.h++

```

1  /*
2  This header is taken quite directly from the C++ standard [2003].
3  According to the standard, a vector is a sequence that supports
4  random-access iterators. More precisely, a vector satisfies all of
5  the requirements of a container, of a reversible container, and of a
6  sequence, including most of the optional sequence requirements,
7  exceptions being the push_front and pop_front functions.
8
9  Authors: Tina A. G. Andersen, Filip Bruman, Jyrki Katajainen, Daniel
10 P. Larsen, Claus Ullerlund, Christian Wolfgang © March 2008
11 */
12
13 #ifndef __CPHSTL_VECTOR__
14 #define __CPHSTL_VECTOR__
15
16 #include <algorithm> // std::equal and std::lexicographical_compare
17 #include <cstddef> // std::size_t and std::ptrdiff_t
18 #include <iterator> // std::reverse_iterator
19 #include <memory> // std::allocator
20 #include <stdexcept> // std::length_error and std::out_of_range
21 #include "type.h++" // cphstl::int2type
22 #include <vector> // std::vector
23
24 namespace cphstl {
25
26     template <
27         typename V,
28         typename A = std::allocator<V>,
29         typename R = std::vector<V, A>,
30         typename I = typename R::iterator,
31         typename J = typename R::const_iterator
32     >
33     class vector {
34     public:
35
36         // types
37
38         typedef V value_type;
39         typedef A allocator_type;
40         typedef typename R::reference reference;
41         typedef typename R::const_reference const_reference;
42         typedef V* pointer;
43         typedef V const* const_pointer;
44         typedef std::ptrdiff_t difference_type;
45         typedef std::size_t size_type;
46         typedef I iterator;
47         typedef J const_iterator;
48         typedef std::reverse_iterator<iterator> reverse_iterator;
49         typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
50
51         // structors
52
53         explicit vector(A const& = A());
54         explicit vector(size_type, V const& = V(), A const& = A());
55
56         template <typename K>
57         vector(K, K, A const& = A());
58
59         vector(vector<V, A, R, I, J> const&);
60         ~vector();

```

```

61     vector<V, A, R, I, J>& operator=(vector<V, A, R, I, J> const&);
62
63     void assign(size_type, V const&);
64
65     template <typename K>
66     void assign(K, K);
67
68     // iterators
69
70     iterator begin();
71     const_iterator begin() const;
72     iterator end();
73     const_iterator end() const;
74     reverse_iterator rbegin();
75     const_reverse_iterator rbegin() const;
76     reverse_iterator rend();
77     const_reverse_iterator rend() const;
78
79     // accessors
80
81     A get_allocator() const;
82     size_type size() const;
83     size_type max_size() const;
84     size_type capacity() const;
85     bool empty() const;
86     const_reference operator[](size_type) const;
87     const_reference at(size_type) const;
88     const_reference front() const;
89     const_reference back() const;
90
91     // modifiers
92
93     reference operator[](size_type);
94     reference at(size_type);
95     reference front();
96     reference back();
97     void resize(size_type, V = V());
98     void reserve(size_type);
99     void push_back(V const&);
100    void pop_back();
101    iterator insert(iterator, V const&);
102    void insert(iterator, size_type, V const&);
103
104    template <typename K>
105    void insert(iterator, K, K);
106
107    iterator erase(iterator);
108    iterator erase(iterator, iterator);
109    void clear();
110    void swap(vector<V, A, R, I, J>&);
111
112    protected:
113
114    template <typename K>
115    void assign_dispatch(K, K, cphstl::int2type<0>);
116
117    template <typename K>
118    void assign_dispatch(K, K, cphstl::int2type<1>);
119
120    void assign_fill(size_type, V const&);
121
122    template <typename K>
123    void assign_range(K, K);
124
125    template <typename K>

```

```

126 void insert_dispatch(iterator, K, K, cphstl::int2type<0>);
127
128 template <typename K>
129 void insert_dispatch(iterator, K, K, cphstl::int2type<1>);
130
131 void insert_fill(iterator, size_type, V const&);
132
133 template <typename K>
134 void insert_range(iterator, K, K);
135
136 typedef R realization_type;
137 realization_type kernel;
138
139 };
140
141 template <typename V, typename A, typename R, typename I, typename J>
142 void swap(vector<V, A, R, I, J>&, vector<V, A, R, I, J>&);
143
144 template <typename V, typename A, typename R, typename I, typename J>
145 bool operator==(vector<V, A, R, I, J> const&, vector<V, A, R, I, J> const&);
146
147 template <typename V, typename A, typename R, typename I, typename J>
148 bool operator<(vector<V, A, R, I, J> const&, vector<V, A, R, I, J> const&);
149
150 template <typename V, typename A, typename R, typename I, typename J>
151 bool operator!=(vector<V, A, R, I, J> const&, vector<V, A, R, I, J> const&);
152
153 template <typename V, typename A, typename R, typename I, typename J>
154 bool operator>(vector<V, A, R, I, J> const&, vector<V, A, R, I, J> const&);
155
156 template <typename V, typename A, typename R, typename I, typename J>
157 bool operator>=(vector<V, A, R, I, J> const&, vector<V, A, R, I, J> const&);
158
159 template <typename V, typename A, typename R, typename I, typename J>
160 bool operator<=(vector<V, A, R, I, J> const&, vector<V, A, R, I, J> const&);
161
162 }
163
164 #include "stl-vector.i++" // implements cphstl::vector
165
166 #endif

```

1.2 Vector/Code/stl-vector.i++

```

1 /*
2  A vector is a bridge that delegates the work to the given
3  realization.
4
5  Authors: Tina A. G. Andersen, Filip Bruman, Jyrki Katajainen, Daniel
6  P. Larsen, Claus Ullerlund, Christian Wolfgang, March 2008
7 */
8
9 namespace cphstl {
10
11     // explicit constructor
12
13     template <typename V, typename A, typename R, typename I, typename J>
14     vector<V, A, R, I, J>::vector(A const& a)
15     : kernel(a) {
16     }
17
18     // parametrized constructors
19
20     template <typename V, typename A, typename R, typename I, typename J>
21     vector<V, A, R, I, J>::vector(size_type s, V const& v, A const& a)

```

```

22     : kernel(a) {
23     (*this).insert((*this).end(), size_type(s), v);
24     }
25
26     template <typename V, typename A, typename R, typename I, typename J>
27     template <typename K>
28     vector<V, A, R, I, J>::vector(K p, K q, A const& a)
29     : kernel(a) {
30     (*this).insert((*this).end(), p, q);
31     }
32
33     // copy constructor
34
35     template <typename V, typename A, typename R, typename I, typename J>
36     vector<V, A, R, I, J>::vector(vector const& other) : kernel(other.get_allocator())
37     {
38     (*this).assign(other.begin(), other.end());
39     }
40
41     // destructor
42
43     template <typename V, typename A, typename R, typename I, typename J>
44     vector<V, A, R, I, J>::~vector() {
45     (*this).clear();
46     }
47
48     // operator=
49
50     template <typename V, typename A, typename R, typename I, typename J>
51     vector<V, A, R, I, J>&::operator=(vector const& other) {
52     (*this).assign(other.begin(), other.end());
53     return *this;
54     }
55
56     // assign
57
58     template <typename V, typename A, typename R, typename I, typename J>
59     void
60     vector<V, A, R, I, J>::assign(size_type n, V const& v) {
61     (*this).assign_fill(n, v);
62     }
63
64     template <typename V, typename A, typename R, typename I, typename J>
65     template <typename K>
66     void
67     vector<V, A, R, I, J>::assign(K p, K q) {
68     (*this).assign_dispatch(p, q, cphstl::int2type<cphstl::type<K>::is_integral>());
69     }
70
71     template <typename V, typename A, typename R, typename I, typename J>
72     template <typename Integer>
73     void
74     vector<V, A, R, I, J>::assign_dispatch(Integer n, Integer v, cphstl::int2type<1>)
75     {
76     (*this).assign_fill(static_cast<size_type>(n), static_cast<V>(v));
77     }
78
79     template <typename V, typename A, typename R, typename I, typename J>
80     template <typename Iterator>
81     void
82     vector<V, A, R, I, J>::assign_dispatch(Iterator p, Iterator q, cphstl::int2type
83     <0>) {
84     (*this).assign_range(p, q);
85     }

```

```

84
85 template <typename V, typename A, typename R, typename I, typename J>
86 void
87 vector<V, A, R, I, J>::assign_fill(size_type n, const W& v) {
88     A a = (*this).kernel.get_allocator();
89     R temporary = R(a);
90     temporary.insert(temporary.begin(), size_type(n), v);
91     (*this).kernel.erase((*this).begin(), (*this).end());
92     (*this).kernel.swap(temporary);
93 }
94
95 template <typename V, typename A, typename R, typename I, typename J>
96 template <typename K>
97 void
98 vector<V, A, R, I, J>::assign_range(K p, K q) {
99     A a = (*this).kernel.get_allocator();
100    R temporary = R(a);
101    temporary.insert(temporary.begin(), p, q);
102    (*this).kernel.erase((*this).begin(), (*this).end());
103    (*this).kernel.swap(temporary);
104 }
105
106 // begin
107
108 template <typename V, typename A, typename R, typename I, typename J>
109 typename vector<V, A, R, I, J>::iterator
110 vector<V, A, R, I, J>::begin() {
111     return iterator((*this).kernel.begin());
112 }
113
114 template <typename V, typename A, typename R, typename I, typename J>
115 typename vector<V, A, R, I, J>::const_iterator
116 vector<V, A, R, I, J>::begin() const {
117     return const_iterator((*this).kernel.begin());
118 }
119
120 // end
121
122 template <typename V, typename A, typename R, typename I, typename J>
123 typename vector<V, A, R, I, J>::iterator
124 vector<V, A, R, I, J>::end() {
125     return (*this).kernel.end();
126 }
127
128 template <typename V, typename A, typename R, typename I, typename J>
129 typename vector<V, A, R, I, J>::const_iterator
130 vector<V, A, R, I, J>::end() const {
131     return (*this).kernel.end();
132 }
133
134 // rbegin
135
136 template <typename V, typename A, typename R, typename I, typename J>
137 typename vector<V, A, R, I, J>::reverse_iterator
138 vector<V, A, R, I, J>::rbegin() {
139     return reverse_iterator((*this).end());
140 }
141
142 template <typename V, typename A, typename R, typename I, typename J>
143 typename vector<V, A, R, I, J>::const_reverse_iterator
144 vector<V, A, R, I, J>::rbegin() const {
145     return const_reverse_iterator((*this).end());
146 }
147
148 // rend

```

```

149
150 template <typename V, typename A, typename R, typename I, typename J>
151 typename vector<V, A, R, I, J>::reverse_iterator
152 vector<V, A, R, I, J>::rend() {
153     return reverse_iterator((*this).begin());
154 }
155
156 template <typename V, typename A, typename R, typename I, typename J>
157 typename vector<V, A, R, I, J>::const_reverse_iterator
158 vector<V, A, R, I, J>::rend() const {
159     return const_reverse_iterator((*this).begin());
160 }
161
162 // get_allocator
163
164 template <typename V, typename A, typename R, typename I, typename J>
165 A
166 vector<V, A, R, I, J>::get_allocator() const {
167     return (*this).kernel.get_allocator();
168 }
169
170 // size
171
172 template <typename V, typename A, typename R, typename I, typename J>
173 typename vector<V, A, R, I, J>::size_type
174 vector<V, A, R, I, J>::size() const {
175     return (*this).kernel.size();
176 }
177
178 // max_size
179
180 template <typename V, typename A, typename R, typename I, typename J>
181 typename vector<V, A, R, I, J>::size_type
182 vector<V, A, R, I, J>::max_size() const {
183     return (*this).kernel.max_size();
184 }
185
186 // capacity
187
188 template <typename V, typename A, typename R, typename I, typename J>
189 typename vector<V, A, R, I, J>::size_type
190 vector<V, A, R, I, J>::capacity() const {
191     return (*this).kernel.capacity();
192 }
193
194 // empty
195
196 template <typename V, typename A, typename R, typename I, typename J>
197 bool
198 vector<V, A, R, I, J>::empty() const {
199     return (*this).size() == size_type(0);
200 }
201
202 // operator[]
203
204 template <typename V, typename A, typename R, typename I, typename J>
205 typename vector<V, A, R, I, J>::reference
206 vector<V, A, R, I, J>::operator[](size_type s) {
207     return (*this).kernel.operator[](s);
208 }
209
210 template <typename V, typename A, typename R, typename I, typename J>
211 typename vector<V, A, R, I, J>::const_reference
212 vector<V, A, R, I, J>::operator[](size_type s) const {
213     return (*this).kernel.operator[](s);

```

```

214 }
215
216 // at
217
218 template <typename V, typename A, typename R, typename I, typename J>
219 typename vector<V, A, R, I, J>::reference
220 vector<V, A, R, I, J>::at(size_type s) {
221     if (s >= (*this).size()) {
222         throw std::out_of_range("index out of bounds");
223     }
224     return (*this).operator[](s);
225 }
226
227 template <typename V, typename A, typename R, typename I, typename J>
228 typename vector<V, A, R, I, J>::const_reference
229 vector<V, A, R, I, J>::at(size_type s) const {
230     if (s >= (*this).size()) {
231         throw std::out_of_range("index out of bounds");
232     }
233     return (*this).operator[](s);
234 }
235
236 // front
237
238 template <typename V, typename A, typename R, typename I, typename J>
239 typename vector<V, A, R, I, J>::reference
240 vector<V, A, R, I, J>::front() {
241     iterator const first = (*this).kernel.begin();
242     return *first;
243 }
244
245 template <typename V, typename A, typename R, typename I, typename J>
246 typename vector<V, A, R, I, J>::const_reference
247 vector<V, A, R, I, J>::front() const {
248     const_iterator const first = (*this).kernel.begin();
249     return *first;
250 }
251
252 // back
253
254 template <typename V, typename A, typename R, typename I, typename J>
255 typename vector<V, A, R, I, J>::reference
256 vector<V, A, R, I, J>::back() {
257     iterator const last = --(*this).end();
258     return *last;
259 }
260
261 template <typename V, typename A, typename R, typename I, typename J>
262 typename vector<V, A, R, I, J>::const_reference
263 vector<V, A, R, I, J>::back() const {
264     const_iterator const last = --(*this).end();
265     return *last;
266 }
267
268 // resize
269
270 template <typename V, typename A, typename R, typename I, typename J>
271 void
272 vector<V, A, R, I, J>::resize(size_type s, V v) {
273     if (s > (*this).size()) {
274         (*this).kernel.insert((*this).end(), size_type(s - (*this).size()), v);
275     }
276     else if (s < (*this).size()) {
277         (*this).kernel.erase((*this).begin() + difference_type(s), (*this).end());
278     }

```

```

279     else {
280         // do nothing
281     }
282 }
283
284 // reserve
285
286 template <typename V, typename A, typename R, typename I, typename J>
287 void
288 vector<V, A, R, I, J>::reserve(size_type s) {
289     (*this).kernel.reserve(s);
290 }
291
292 // push_back
293
294 template <typename V, typename A, typename R, typename I, typename J>
295 void
296 vector<V, A, R, I, J>::push_back(V const& v) {
297     (*this).kernel.insert((*this).end(), v);
298 }
299
300 // pop_back
301
302 template <typename V, typename A, typename R, typename I, typename J>
303 void
304 vector<V, A, R, I, J>::pop_back() {
305     iterator last = --(*this).end();
306     (*this).kernel.erase(last);
307 }
308
309 // single-element insert
310
311 template <typename V, typename A, typename R, typename I, typename J>
312 typename vector<V, A, R, I, J>::iterator
313 vector<V, A, R, I, J>::insert(iterator p, V const& v) {
314     return (*this).kernel.insert(p, v);
315 }
316
317 // multiple-element inserts
318
319 template <typename V, typename A, typename R, typename I, typename J>
320 void
321 vector<V, A, R, I, J>::insert(iterator p, size_type n, V const& v) {
322     (*this).insert_fill(p, n, v);
323 }
324
325 template <typename V, typename A, typename R, typename I, typename J>
326 template <typename K>
327 void
328 vector<V, A, R, I, J>::insert(iterator p, K q, K r) {
329     (*this).insert_dispatch(p, q, r, cphstl::int2type<cphstl::type<K>::is_integral
330 >());
331 }
332
333 template <typename V, typename A, typename R, typename I, typename J>
334 template <typename K>
335 void
336 vector<V, A, R, I, J>::insert_dispatch(iterator p, K n, K v, cphstl::int2type<I>)
337 {
338     (*this).insert_fill(p, static_cast<size_type>(n), static_cast<V>(v));
339 }
340
341 template <typename V, typename A, typename R, typename I, typename J>
342 template <typename K>
343 void

```

```

342 vector<V, A, R, I, J>::insert_dispatch(iterator p, K q, K r, cphstl::int2type<0>)
    {
343     (*this).insert_range(p, q, r);
344 }
345
346 template <typename V, typename A, typename R, typename I, typename J>
347 void
348 vector<V, A, R, I, J>::insert_fill(iterator p, size_type s, V const& v) {
349     (*this).kernel.insert(p, s, v);
350 }
351
352 template <typename V, typename A, typename R, typename I, typename J>
353 template <typename K>
354 void
355 vector<V, A, R, I, J>::insert_range(iterator p, K q, K r) {
356     (*this).kernel.insert(p, q, r);
357 }
358
359 // single-element erase
360
361 template <typename V, typename A, typename R, typename I, typename J>
362 typename vector<V, A, R, I, J>::iterator
363 vector<V, A, R, I, J>::erase(iterator p) {
364     return (*this).kernel.erase(p);
365 }
366
367 // multiple-element erase
368
369 template <typename V, typename A, typename R, typename I, typename J>
370 typename vector<V, A, R, I, J>::iterator
371 vector<V, A, R, I, J>::erase(iterator p, iterator q) {
372     return (*this).kernel.erase(p, q);
373 }
374
375 // clear
376
377 template <typename V, typename A, typename R, typename I, typename J>
378 void
379 vector<V, A, R, I, J>::clear() {
380     (*this).kernel.erase((*this).begin(), (*this).end());
381 }
382
383 // swap
384
385 template <typename V, typename A, typename R, typename I, typename J>
386 void
387 vector<V, A, R, I, J>::swap(vector<V, A, R, I, J>& r) {
388     (*this).kernel.swap(r.kernel);
389 }
390
391 template <typename V, typename A, typename R, typename I, typename J>
392 void
393 swap(vector<V, A, R, I, J>& r, vector<V, A, R, I, J>& s) {
394     r.swap(s);
395 }
396
397 // operator==
398
399 template <typename V, typename A, typename R, typename I, typename J>
400 bool
401 operator==(vector<V, A, R, I, J> const& r, vector<V, A, R, I, J> const& s) {
402     return (r.size() == s.size() && std::equal(r.begin(), r.end(), s.begin()));
403 }
404
405 // operator<

```

```

406
407 template <typename V, typename A, typename R, typename I, typename J>
408 bool
409 operator<(vector<V, A, R, I, J> const& r, vector<V, A, R, I, J> const& s) {
410     return std::lexicographical_compare(r.begin(), r.end(), s.begin(), s.end());
411 }
412
413 // operator!=
414
415 template <typename V, typename A, typename R, typename I, typename J>
416 bool
417 operator!=(vector<V, A, R, I, J> const& r, vector<V, A, R, I, J> const& s) {
418     return !(r == s);
419 }
420
421 // operator>
422
423 template <typename V, typename A, typename R, typename I, typename J>
424 bool
425 operator>(vector<V, A, R, I, J> const& r, vector<V, A, R, I, J> const& s) {
426     return (s < r);
427 }
428
429 // operator>=
430
431 template <typename V, typename A, typename R, typename I, typename J>
432 bool
433 operator>=(vector<V, A, R, I, J> const& r, vector<V, A, R, I, J> const& s) {
434     return !(r < s);
435 }
436
437 // operator<=
438
439 template <typename V, typename A, typename R, typename I, typename J>
440 bool
441 operator<=(vector<V, A, R, I, J> const& r, vector<V, A, R, I, J> const& s) {
442     return !(s < r);
443 }
444
445 }

```

2. Framework

2.1 Vector-framework/Code/vector-framework.h++

```

1 #ifndef __CPHSTL_VECTOR_FRAMEWORK__
2 #define __CPHSTL_VECTOR_FRAMEWORK__
3
4 /*
5  A skeleton of a vector implementation. Functionality is put into
6  kernels and encapsulators.
7
8  Authors: Tina A. G. Andersen, Ulrik Schou Jørgensen, Jyrki
9  Katajainen, Bo Simonsen, Mikkel Jønsson Thomsen, Claus Ullerlund,
10 Bue Krogh Vedel-Larsen © 2008, 2009
11 */
12
13 #include <cassert>
14 #include <stddef> // std::size_t and std::ptrdiff_t
15 #include <iostream>
16 #include <memory> // std::allocator
17 #include <stdexcept> // std::out_of_range
18 #include <utility> // std::pair
19

```

```

20 #include "allocator-proxy.h++"
21 #include "dynamic-array.h++"
22 #include "encapsulator-factory.h++"
23 #include "general-surrogate.h++"
24 #include "reference-proxy.h++"
25 #include "slot-swap.i++"
26
27 namespace cphstl {
28
29     template <
30         typename V,
31         typename A = std::allocator<V>,
32         typename K = dynamic_array<V, A>
33     >
34     class vector_framework {
35
36     public:
37
38         // types
39
40         typedef V value_type;
41         typedef A allocator_type;
42         typedef K kernel_type;
43         typedef std::size_t size_type;
44         typedef general_surrogate<K> surrogate_type;
45         typedef typename kernel_type::encapsulator_type encapsulator_type;
46         typedef std::pair<size_type, surrogate_type*> concrete_iterator;
47         typedef std::pair<size_type, surrogate_type*> concrete_const_iterator;
48         typedef reference_proxy<V, A, false, encapsulator_type> reference;
49         typedef reference_proxy<V, A, true, encapsulator_type> const_reference;
50         typedef encapsulator_factory<V, A, encapsulator_type> factory_type;
51
52         // structs
53
54         explicit vector_framework(A const& = A());
55         virtual ~vector_framework();
56
57         // iterators
58
59         concrete_iterator end();
60         concrete_const_iterator end() const;
61         concrete_iterator begin();
62         concrete_const_iterator begin() const;
63
64         // accessors
65
66         A get_allocator() const;
67         size_type size() const;
68         size_type max_size() const;
69         size_type capacity() const;
70         const_reference operator [] (size_type) const;
71
72         // modifiers
73
74         reference operator [] (size_type);
75         void reserve(size_type);
76         concrete_iterator insert(concrete_iterator, V const&);
77         void insert(concrete_iterator, size_type, V const&);
78
79         template <typename IT>
80         void insert(concrete_iterator, IT, IT);
81
82         concrete_iterator erase(concrete_iterator);
83         concrete_iterator erase(concrete_iterator, concrete_iterator);
84         void swap(vector_framework<V, A, K>&);

```

```

85
86 protected:
87
88     size_type erase_values(concrete_iterator pos, size_type s);
89     void block_copy_backward(size_type, size_type, size_type);
90     void block_copy_forward(size_type, size_type, size_type);
91
92 private:
93
94     typedef typename A::template rebind<surrogate_type>::other
95         surrogate_allocator_type;
96
97     allocator_proxy<surrogate_allocator_type> surrogate_allocator;
98     factory_type f;
99     A a;
100    K k;
101    surrogate_type* s;
102 };
103
104 #include "vector-framework.i++"
105 #endif

```

2.2 Vector-framework/Code/vector-framework.i++

```

1 /*
2  An implementation of cphstl::vector-framework
3
4  Authors: Tina A. G. Andersen, Ulrik Schou Jørgensen, Jyrki
5 Katajainen, Mads D. Kristensen, Bo Simonsen, Mikkel Jønsson Thomsen,
6 Claus Ullerlund, Bue Krogh Vedel-Larsen © 2008, 2009
7 */
8
9 #include <algorithm> // std::swap
10 #include <cassert> // assert
11
12 namespace cphstl {
13
14     // constructor
15
16     template <typename V, typename A, typename K>
17     vector_framework<V, A, K>::vector_framework(A const& _a)
18     : surrogate_allocator(_a), a(_a), f(_a), k(_a)
19     {
20         (*this).s = (*this).surrogate_allocator.allocate(1);
21         ((*this).s).subject() = &(*this).k;
22     }
23
24     // destructor
25
26     template <typename V, typename A, typename K>
27     vector_framework<V, A, K>::~~vector_framework() {
28         (*this).surrogate_allocator.deallocate((*this).s, 1);
29     }
30
31     // begin
32
33     template <typename V, typename A, typename K>
34     typename vector_framework<V, A, K>::concrete_iterator
35     vector_framework<V, A, K>::begin() {
36         return concrete_iterator(0, (*this).s);
37     }
38
39     template <typename V, typename A, typename K>
40     typename vector_framework<V, A, K>::concrete_const_iterator

```

```

41 vector_framework<V, A, K>::begin() const {
42     return concrete_const_iterator(0, (*this).s);
43 }
44
45 // end
46
47 template <typename V, typename A, typename K>
48 typename vector_framework<V, A, K>::concrete_iterator
49 vector_framework<V, A, K>::end() {
50     return concrete_iterator((*this).k.size(), (*this).s);
51 }
52
53 template <typename V, typename A, typename K>
54 typename vector_framework<V, A, K>::concrete_const_iterator
55 vector_framework<V, A, K>::end() const {
56     return concrete_const_iterator((*this).k.size(), (*this).s);
57 }
58
59 // get_allocator
60 //
61 // Return the allocator
62 template <typename V, typename A, typename K>
63 A
64 vector_framework<V, A, K>::get_allocator() const {
65     return A((*this).a);
66 }
67
68 // size
69 //
70 // Return the number of elements stored
71 template <typename V, typename A, typename K>
72 typename vector_framework<V, A, K>::size_type
73 vector_framework<V, A, K>::size() const {
74     return (*this).k.size();
75 }
76
77 // max_size
78 //
79 // Return the maximum number of elements possible
80 template <typename V, typename A, typename K>
81 typename vector_framework<V, A, K>::size_type
82 vector_framework<V, A, K>::max_size() const {
83     return (*this).k.max_size();
84 }
85
86 // capacity
87 //
88 // Return the maximum number of elements without reallocation
89 template <typename V, typename A, typename K>
90 typename vector_framework<V, A, K>::size_type
91 vector_framework<V, A, K>::capacity() const {
92     return (*this).k.capacity();
93 }
94
95 // operator[] const
96 //
97 // Return a reference to the immutable element with index s
98 template <typename V, typename A, typename K>
99 typename vector_framework<V, A, K>::const_reference
100 vector_framework<V, A, K>::operator[](size_type s) const {
101     return reference_proxy<V, A, true, typename K::encapsulator_type>((*this).k.
        access(s));
102 }
103
104 // operator[]

```

```

105 //
106 // Return a reference to the element with index s
107 template <typename V, typename A, typename K>
108 typename vector_framework<V, A, K>::reference
109 vector_framework<V, A, K>::operator[](size_type s) {
110     return reference_proxy<V, A, false, typename K::encapsulator_type>((*this).k.
        access(s));
111 }
112
113 // reserve
114 //
115 // Enlarge the capacity, if not large enough
116 template <typename V, typename A, typename K>
117 void
118 vector_framework<V, A, K>::reserve(size_type s) {
119     (*this).k.grow(s);
120 }
121
122 // insert
123 //
124 // Insert a copy of v before position p; return its position
125 template <typename V, typename A, typename K>
126 typename vector_framework<V, A, K>::concrete_iterator
127 vector_framework<V, A, K>::insert(concrete_iterator pos,
        value_type const& v) {
128     if ((*this).size() == (*this).capacity()) {
129         (*this).k.grow(1);
130     }
131 }
132
133     (*this).f.create(v, (*this).k.access((*this).k.size()), (*this).k.size());
134
135     if (pos.first != (*this).k.size()) {
136         (*this).block_copy_backward(pos.first, (*this).k.size(), 1);
137     }
138
139     (*this).k.size((*this).k.size() + 1);
140
141     return concrete_iterator(pos.first, (*this).s);
142 }
143
144 // insert
145 //
146 // Insert s copies of v before position p; return nothing
147 template <typename V, typename A, typename K>
148 void
149 vector_framework<V, A, K>::insert(concrete_iterator pos, size_type s,
        value_type const& v) {
150     size_type p = pos.first;
151
152     if ((*this).size() + s >= (*this).capacity()) {
153         (*this).k.grow(s);
154     }
155
156     for (size_type i = 0; i < s; ++i) {
157         (*this).f.create(v, (*this).k.access((*this).k.size() + i), (*this).k.size() +
            i);
158     }
159
160     if (p != (*this).k.size()) {
161         (*this).block_copy_backward(p, (*this).k.size(), s);
162     }
163
164     (*this).k.size((*this).k.size() + s);
165 }
166
167

```

```

168 // insert
169
170 template <typename V, typename A, typename K>
171 template <typename IT>
172 void
173 vector_framework<V, A, K>::insert(concrete_iterator p, IT q, IT r) {
174
175     size_type old_size = (*this).k.size();
176
177     IT iter = q;
178     while (iter != r){
179
180         if ((*this).size() == (*this).capacity()) {
181             (*this).k.grow(1);
182         }
183
184         (*this).f.create(*iter, (*this).k.access((*this).k.size()), (*this).k.size());
185
186         p.first++;
187         ++iter;
188         (*this).k.size((*this).k.size() + 1);
189     }
190
191     if (p.first != (*this).k.size()) {
192         (*this).block_copy_backward(p.first, (*this).k.size(), (*this).k.size() -
193             old_size);
194     }
195 }
196
197 // erase
198 //
199 // Remove the element at position p; return the position of the next element
200 template <typename V, typename A, typename K>
201 typename vector_framework<V, A, K>::concrete_iterator
202 vector_framework<V, A, K>::erase(concrete_iterator pos) {
203
204     size_type index = (*this).erase_values( pos, size_type( 1 ) );
205     return concrete_iterator(index, (*this).s);
206 }
207
208 // erase
209 //
210 // Remove all elements in the range [p,q); return q
211 template <typename V, typename A, typename K>
212 typename vector_framework<V, A, K>::concrete_iterator
213 vector_framework<V, A, K>::erase(concrete_iterator pos_p,
214     concrete_iterator pos_q) {
215
216     size_type range = pos_q.first - pos_p.first;
217     size_type index = (*this).erase_values( pos_p, range );
218
219     return concrete_iterator(index, (*this).s);
220 }
221
222 // swap
223 //
224 // Swap the data of the calling dynamic array and r
225 template <typename V, typename A, typename K>
226 void
227 vector_framework<V, A, K>::swap(vector_framework<V, A, K>& r) {
228     std::swap((*this).s.subject(), (*r.s).subject());
229     std::swap((*this).s, r.s);
230     std::swap((*this).k, r.k);
231     std::swap((*this).f, r.f);

```

```

232 }
233
234 // erase_values
235 //
236 // Erase s entries from the array
237 template <typename V, typename A, typename K>
238 typename vector_framework<V, A, K>::size_type
239 vector_framework<V, A, K>::erase_values(concrete_iterator pos, size_type s) {
240
241     size_type p = pos.first;
242
243     if (s == 0) {
244         return p;
245     }
246
247     if (p+s != (*this).k.size()) {
248         (*this).block_copy_forward(p, (*this).k.size(), s);
249     }
250
251     for (size_type i = (*this).k.size() - s; i < (*this).k.size(); ++i) {
252         (*this).f.destroy((*this).k.access(i));
253     }
254
255     (*this).k.size((*this).k.size() - s);
256
257     if ((*this).capacity() != 0) {
258         (*this).k.shrink();
259     }
260
261     return p;
262 }
263
264 template <typename V, typename A, typename K>
265 void
266 vector_framework<V, A, K>::block_copy_backward(size_type start, size_type end,
267     size_type num) {
268
269     size_t s = num;
270
271     for (size_type i = end + s - 1; i >= start + s; --i) {
272         slot_swap((*this).k.access(i), (*this).k.access(i - s));
273     }
274 }
275
276 template <typename V, typename A, typename K>
277 void
278 vector_framework<V, A, K>::block_copy_forward(size_type start, size_type end,
279     size_type num) {
280
281     if (num >= end)
282         return;
283
284     size_t s = num;
285
286     for (size_type i = start; i < end - 1; ++i) {
287         slot_swap((*this).k.access(i), (*this).k.access(i + s));
288     }
289 }

```

3. Encapsulators

3.1 *Vector-framework/Code/direct-encapsulator.h++*

```

1 #ifndef __CPHSTL_DIRECT_ENCAPSULATOR__
2 #define __CPHSTL_DIRECT_ENCAPSULATOR__
3
4 /*
5  A direct encapsulator encapsulates an element, nothing else.
6
7  Author: Bo Simonsen © 2008, 2009
8 */
9
10 #include <cstddef> // defines std::size_t and std::ptrdiff_t
11
12 namespace cphstl {
13
14     template <typename V, typename A>
15     class direct_encapsulator {
16
17     public:
18         typedef V value_type;
19         typedef direct_encapsulator slot_type;
20         typedef direct_encapsulator* segment_type;
21         enum {size = sizeof(value_type) };
22
23         direct_encapsulator(value_type const& _v) : v(_v) {
24         }
25
26         direct_encapsulator(direct_encapsulator const& a) : v(a.v) {
27         }
28
29         const value_type& content() const {
30             return v;
31         }
32
33         value_type& content() {
34             return v;
35         }
36
37     private:
38         value_type v;
39     };
40 }
41 #endif

```

3.2 *Vector-framework/Code/indirect-encapsulator.h++*

```

1 #ifndef __CPHSTL_INDIRECT_ENCAPSULATOR__
2 #define __CPHSTL_INDIRECT_ENCAPSULATOR__
3
4 /*
5  Defines an indirect encapsulator.
6  Segments store pointers to objects
7  of the class indirect_encapsulator which store the value.
8
9  Authors: Bo Simonsen, May 2008, September 2008
10  Bue Krogh Vedel-Larsen and Mikkel Jønsson Thomsen, March 2009
11 */
12
13 #include <cassert>
14 #include <memory.h>
15
16 #include "allocator-proxy.h++"

```

```

17
18 namespace cphstl {
19   template <typename V, typename A>
20   class indirect_encapsulator;
21
22   template <typename V, typename A>
23   void slot_swap(indirect_encapsulator<V, A>*&, indirect_encapsulator<V, A>*&);
24
25   template <typename V, typename A>
26   class indirect_encapsulator {
27   private:
28     typedef indirect_encapsulator<V, A> this_type;
29     typedef typename A::template rebind<this_type>::other input_allocator_type;
30
31   public:
32     friend void slot_swap<>(indirect_encapsulator<V, A>*&, indirect_encapsulator<V,
33                             A>*&);
34
35     typedef V value_type;
36     typedef A allocator_type;
37     typedef std::size_t size_type;
38     typedef this_type** segment_type;
39     typedef this_type* slot_type;
40
41     enum {size = sizeof(this_type*) +
42           sizeof(size_type) +
43           sizeof(value_type)};
44   public:
45     ~indirect_encapsulator() {}
46     indirect_encapsulator(value_type const& _v, input_allocator_type& _a, size_type
47                           const& _index) {
48       (*this).v = _v;
49       (*this).p = _index;
50     }
51     value_type & content() {
52       return (*this).v;
53     }
54     value_type const& content() const {
55       return (*this).v;
56     }
57
58     size_type position() const {
59       return (*this).p;
60     }
61
62   private:
63     value_type v;
64     size_type p;
65
66 };
67 }
68 }
69
70 #endif // _CPHSTLSPDA.SP.DYNAMICARRAY.ENTRY.H

```

3.3 Vector-framework/Code/doubly-indirect-encapsulator.h++

```

1 #ifndef __CPHSTL_DOUBLY_INDIRECT_ENCAPSULATOR__
2 #define __CPHSTL_DOUBLY_INDIRECT_ENCAPSULATOR__
3
4 /*
5  Defines a doubly indirect encapsulator.
6  Segments store pointers to objects

```

```

7  of the class doubly_indirect_encapsulator. Each
8  object of doubly_indirect_encapsulator store a
9  pointer to an object of the class value_encapsulator
10 which stores the value.
11
12 Authors: Bo Simonsen, May 2008, September 2008
13 Bue Krogh Vedel-Larsen and Mikkel Jønsson Thomsen, March 2009
14 */
15
16 #include <cassert>
17 #include <memory.h>
18
19 namespace cphstl {
20
21     template <typename V>
22     class value_encapsulator {
23
24     public:
25         typedef V value_type;
26
27         value_encapsulator(value_type const& value)
28             : value(value) {
29         }
30
31         value_type& content() {
32             return value;
33         }
34
35         value_type const& content() const {
36             return value;
37         }
38
39     private:
40         value_type value;
41     };
42
43     template <typename V, typename A>
44     class doubly_indirect_encapsulator;
45
46     template <typename V, typename A>
47     void slot_swap(doubly_indirect_encapsulator<V, A>* & a, doubly_indirect_encapsulator<
48         V, A>* & b);
49
50     template <typename V, typename A, bool is_const, typename E>
51     class reference_proxy;
52
53     template <typename V, typename A>
54     class doubly_indirect_encapsulator {
55     private:
56         typedef doubly_indirect_encapsulator<V, A> this_type;
57         typedef typename A::template
58             rebind<this_type>::other input_allocator_type;
59     public:
60         typedef V value_type;
61         typedef A allocator_type;
62         typedef std::size_t size_type;
63         typedef this_type** segment_type;
64         typedef this_type* slot_type;
65         typedef value_encapsulator<V> value_encapsulator_type;
66         typedef typename A::template
67             rebind<value_encapsulator_type>::other value_encapsulator_allocator_type;
68
69     enum {size = sizeof(this_type*) +
70             sizeof(value_encapsulator_type*) +
71             sizeof(size_type) +

```

```

71         sizeof(value_type)};
72
73     friend void slot_swap<>(doubly_indirect_encapsulator<V, A>*&,
74         doubly_indirect_encapsulator<V, A>*&);
75     friend class reference_proxy<V, A, false, this_type>;
76     friend class reference_proxy<V, A, true, this_type >;
77
78 public:
79     doubly_indirect_encapsulator(value_type const& _v, input_allocator_type& _a,
80         size_type _index) {
81         if ((*this).ap == 0) {
82             (*this).a = _a;
83             (*this).ap = &(*this).a;
84         }
85
86         (*this).v = (*this).a.allocate(1);
87         new ((*this).v) value_encapsulator_type(_v);
88         (*this).p = _index;
89     }
90
91     ~doubly_indirect_encapsulator() {
92         ((*this).v).~value_encapsulator_type();
93         (*this).a.deallocate((*this).v, 1);
94     }
95
96     value_type& content() {
97         return (*this).v->content();
98     }
99
100    value_type const& content() const {
101        return (*this).v->content();
102    }
103
104    size_type position() const {
105        return (*this).p;
106    }
107
108 private:
109     static value_encapsulator_allocator_type a;
110     static value_encapsulator_allocator_type* ap;
111     size_type p;
112     value_encapsulator_type* v;
113 };
114
115 template <typename V, typename A>
116 typename doubly_indirect_encapsulator<V, A>::value_encapsulator_allocator_type*
117     doubly_indirect_encapsulator<V, A>::ap = 0;
118
119 template <typename V, typename A>
120 typename doubly_indirect_encapsulator<V, A>::value_encapsulator_allocator_type
121     doubly_indirect_encapsulator<V, A>::a;
122
123 #endif

```

4. Encapsulator factories

4.1 Vector-framework/Code/encapsulator-factory.h++

```

1 #ifndef __CPHSTL_ENCAPSULATOR_FACTORY__
2 #define __CPHSTL_ENCAPSULATOR_FACTORY__
3
4 /*
5  The factory class templates. We provide two versions of this class,

```

```

6  a general version which is used to create indirect/doubly-indirect
7  encapsulators and a specialized version for the direct encapsulator.
8  These classes ensure that the creation of objects is done in a
9  generic way with respect to the framework class.
10
11  Authors: Jyrki Katajainen, Bo Simonsen © 2008
12  */
13
14  #include "allocator-proxy.h++"
15
16  namespace cphstl {
17
18  template <typename V, typename A>
19  class direct_encapsulator;
20
21  template <typename V, typename A, typename E>
22  class encapsulator_factory {
23  public:
24
25      typedef E encapsulator_type;
26      typedef typename encapsulator_type::value_type value_type;
27      typedef typename A::template rebind<encapsulator_type>::other
28          encapsulator_allocator_type;
29      typedef std::size_t size_type;
30
31      encapsulator_factory(A const& a = A()) : encapsulator_allocator(a) {
32      }
33
34      encapsulator_factory(encapsulator_factory const& f) : encapsulator_allocator(f.
35          encapsulator_allocator) {
36      }
37
38      void create(value_type const& v, encapsulator_type*& e, size_type const& index)
39      {
40          e = encapsulator_allocator.allocate(1);
41          new(e) encapsulator_type(v, encapsulator_allocator.subject(), index);
42      }
43
44      void destroy(encapsulator_type* entry) {
45          (*entry).~encapsulator_type();
46          encapsulator_allocator.deallocate(entry, 1);
47      }
48
49  private:
50
51      allocator_proxy<encapsulator_allocator_type> encapsulator_allocator;
52  };
53
54  template <typename V, typename A>
55  class encapsulator_factory <V, A, direct_encapsulator<V, A> > {
56  public:
57
58      typedef direct_encapsulator<V, A> encapsulator_type;
59      typedef typename encapsulator_type::value_type value_type;
60      typedef std::size_t size_type;
61
62      encapsulator_factory(A const& a = A()) {
63      }
64
65      void create(value_type const& v, encapsulator_type& e, size_type const& index) {
66          new (&e) encapsulator_type(v);
67      }
68
69      void destroy(encapsulator_type& e) {
70          e.~encapsulator_type();
71      }
72  };

```

```

68     }
69   };
70 };
71 }
72 #endif

```

5. General surrogate

5.1 Vector-framework/Code/general-surrogate.h++

```

1 #ifndef __CPHSTL_GENERAL_SURROGATE__
2 #define __CPHSTL_GENERAL_SURROGATE__
3
4 /*
5  The surrogate keeps a pointer to some subject. For example, in the
6  vector framework an iterator keeps a pointer to a kernel surrogate.
7  This is done in order to maintain referential integrity when
8  swapping two vectors.
9
10 Author: Bo Simonsen © 2009
11 */
12
13 namespace cphstl {
14
15   template <typename S>
16   class general_surrogate {
17   public:
18
19     typedef S subject_type;
20
21     subject_type*& subject() {
22       return ptr;
23     }
24
25   private:
26
27     subject_type* ptr;
28   };
29 }
30
31 #endif

```

6. Other proxies

6.1 Vector-framework/Code/reference-proxy.h++

```

1 #ifndef __CPHSTL_REFERENCE_PROXY__
2 #define __CPHSTL_REFERENCE_PROXY__
3
4 /*
5  Defines the reference proxy. The framework will, instead
6  of returning references, return an object of the reference
7  proxy.
8
9  Author: Bo Simonsen © 2009
10 */
11
12 #include "type.h++" // defines cphstl::if_then_else
13
14 namespace cphstl {
15
16   template <typename V, typename A>
17   class doubly_indirect_encapsulator;

```

```

18
19 template <typename V, typename A>
20 class indirect_encapsulator;
21
22 template <typename V, typename A, bool is_const, typename E>
23 class reference_proxy {
24
25 public:
26     typedef typename E::value_type value_type;
27     typedef typename cphstl::if_then_else<is_const, E const&, E&>::type
        encapsulator_reference;
28     typedef typename cphstl::if_then_else<is_const, E const*, E*>::type
        encapsulator_pointer;
29     typedef typename cphstl::if_then_else<is_const, value_type const&, value_type
        &>::type reference_type;
30
31     reference_proxy(value_type& _v) : e(0), v(_v) {
32     }
33
34     reference_proxy(value_type const& _v) : e(0), v(_v) {
35     }
36
37     reference_proxy(encapsulator_reference _e) : e(&_e), v() {
38     }
39
40     reference_proxy(reference_proxy const& r) : e(r.e), v() {
41     }
42
43     value_type const& operator==(value_type const& _v) {
44         if (e == 0) {
45             (*this).v = _v;
46             return (*this).v;
47         }
48         (*this).e->content() = _v;
49         return (*this).e->content();
50     }
51
52     operator reference_type() {
53         if (e == 0) {
54             return v;
55         }
56         return (*this).e->content();
57     }
58
59 private:
60     encapsulator_pointer e;
61     V v;
62 };
63
64 template <typename V, typename A, bool is_const>
65 class reference_proxy<V, A, is_const, indirect_encapsulator<V, A>> {
66
67 public:
68     typedef indirect_encapsulator<V, A> entry_type;
69     typedef typename entry_type::value_type value_type;
70     typedef typename cphstl::if_then_else<is_const, entry_type const*, entry_type
        *>::type entry_ptr;
71     typedef typename cphstl::if_then_else<is_const, value_type const&, value_type
        &>::type reference_type;
72
73     reference_proxy(value_type& _v) : e(0), v(_v) {
74     }
75
76     reference_proxy(value_type const& _v) : e(0), v(_v) {
77     }

```

```

78
79     reference_proxy(entry_ptr _e) : e(_e) {
80     }
81
82     reference_proxy(reference_proxy const& r) : e(r.e), v(r.v) {
83     }
84
85     value_type const& operator=(value_type const& _v) {
86         if (e == 0) {
87             (*this).v = _v;
88             return (*this).v;
89         }
90         (*this).e->content() = _v;
91         return (*this).e->content();
92     }
93
94     operator reference_type() {
95         if ((*this).e == 0) {
96             return v;
97         }
98         return (*this).e->content();
99     }
100
101 private:
102     entry_ptr e;
103     V v;
104
105 };
106
107 template <typename V, typename A, bool is_const>
108 class reference_proxy<V, A, is_const, doubly_indirect_encapsulator<V, A> > {
109
110 public:
111     typedef doubly_indirect_encapsulator<V, A> encapsulator_type;
112     typedef typename encapsulator_type::value_type value_type;
113     typedef typename encapsulator_type::value_encapsulator_type
114         value_encapsulator_type;
115     typedef typename A::template rebind<value_encapsulator_type>::other
116         allocator_type;
117     typedef typename cphstl::if_then_else<is_const, encapsulator_type const*,
118         encapsulator_type*>::type encapsulator_pointer;
119     typedef typename cphstl::if_then_else<is_const, value_type const&, value_type
120         &>::type reference_type;
121
122     reference_proxy(encapsulator_pointer _e) : e(_e) {
123     }
124
125     reference_proxy(value_type& _v) : v(_v), e(0) {
126     }
127
128     reference_proxy(value_type const& _v) : v(_v), e(0) {
129     }
130
131     reference_proxy(reference_proxy const& r) : e(r.e) {
132     }
133
134     operator reference_type() {
135         if ((*this).e == 0) {
136             return (*this).v;
137         }
138         return (*this).e->content();
139     }
140
141     value_type const& operator=(value_type const& _v) {
142         value_encapsulator_type* new_v;

```

```

139     value_encapsulator_type* old_v;
140
141     try {
142         new_v = (*this).allocator.allocate(1);
143         new (new_v) value_encapsulator_type(_v);
144
145         old_v = (*(this).e).v;
146         (*(this).e).v = new_v;
147
148         (*old_v).~value_encapsulator_type();
149         (*this).allocator.deallocate(old_v, 1);
150     }
151     catch (...) {
152         (*this).allocator.deallocate(new_v, 1);
153         throw;
154     }
155
156     return _v;
157 }
158
159 reference_proxy& operator=(reference_proxy const& r) {
160     (*this).e = r.e;
161     return (*this);
162 }
163
164 private:
165     allocator_type allocator;
166     encapsulator_pointer e;
167     value_type v;
168 };
169
170 }
171 #endif

```

6.2 Proxy/Code/allocator-proxy.h++

```

1 #ifndef __CPHSTL_ALLOCATOR_PROXY__
2 #define __CPHSTL_ALLOCATOR_PROXY__
3
4 /*
5  A generic allocator proxy class. It is actually more a real
6  encapsulator class as described in [1]. It keeps an instance
7  of the allocator, given by the parameterized constructor,
8  as a pointer. The allocator instance is allocated. The purpose of
9  this class is to perform a safe swap operation for all containers.
10 Swap must not throw an exception, so when swapping this proxy class
11 we swap pointers which cannot fail.
12 All STL containers keeps an allocator which makes this class generic.
13
14 [1] Pascoe, Geoffrey A. Encapsulators: a new software paradigm in
15 Smalltalk-80. SIGPLAN Notices 21(11), 1986, 341-346.
16
17 Author: Bo Simonsen © 2009
18 */
19
20 #include <algorithm>
21
22 namespace cphstl {
23     template <typename A>
24     class allocator_proxy {
25     public:
26         typedef typename A::size_type size_type;
27         typedef typename A::difference_type difference_type;
28         typedef typename A::pointer pointer;

```

```

29     typedef typename A::const_pointer const_pointer;
30     typedef typename A::reference reference;
31     typedef typename A::const_reference const_reference;
32     typedef typename A::value_type value_type;
33
34     template <class U>
35     struct rebind { typedef allocator_proxy<typename A::template rebind<U>::other >
36         other; };
37
38     allocator_proxy() {
39         (*this).a = new A();
40     }
41     allocator_proxy(A const& a) {
42         (*this).a = new A(a);
43     }
44     ~allocator_proxy() {
45         delete (*this).a;
46     }
47     allocator_proxy(allocator_proxy const& ap) {
48         (*this).a = new A(*ap.a);
49     }
50     allocator_proxy(allocator_proxy& ap) {
51         (*this).a = new A(*ap.a);
52     }
53     allocator_proxy operator=(allocator_proxy const& ap) {
54         delete (*this).a;
55         (*this).a = new A(*ap.a);
56         return (*this);
57     }
58     allocator_proxy operator=(A const& a) {
59         delete (*this).a;
60         (*this).a = new A(a);
61         return (*this);
62     }
63     pointer address(reference ref) const {
64         return (*a).address(ref);
65     }
66     const_pointer address(const_reference const_ref) const {
67         return (*a).address(const_ref);
68     }
69     size_type max_size() const throw() {
70         return (*a).max_size();
71     }
72     pointer allocate(size_type s, const_pointer p = 0) {
73         return (*a).allocate(s);
74     }
75     void construct(pointer p, value_type const& v) {
76         (*a).construct(p,v);
77     }
78     void destroy(pointer p){
79         (*a).construct(p);
80     }
81     void deallocate(pointer p, size_type s) {
82         (*a).deallocate(p,s);
83     }
84     A& subject() const {
85         return *a;
86     }
87 private:
88     A* a;
89 };
90
91 }
92

```

```
93 #endif
```

7. Iterators

7.1 Iterator/Code/rank-iterator.h++

```
1  /*
2  A rank-based iterator is just a (pointer, index) pair where the
3  pointer points to the data structure containing the cell referred to
4  and the index is the rank of that cell in the sequence of cells
5  storing the elements.
6
7  The idea of combining iterators and const iterators into the same
8  class is taken from [Matt Austern. Defining iterators and const
9  iterators. C/C++ User's Journal 19,1 (2001), 74-79].
10
11 Authors: Jyrki Katajainen, Bo Simonsen © 2008
12 */
13
14 #ifndef __CPHSTL_RANK_ITERATOR__
15 #define __CPHSTL_RANK_ITERATOR__
16
17 #include <cstddef> // defines std::size_t and std::ptrdiff_t
18 #include <iterator> // defines std::random_access_iterator_tag
19 #include "type.h++" // defines cphstl::if_then_else
20 #include <utility> // defines std::pair
21
22 namespace cphstl {
23
24     template <typename V, typename A, typename R, typename I, typename J>
25     class vector;
26
27     template <typename R, bool is_const = false, typename E = typename R::
28         encapsulator_type>
29     class rank_iterator {
30     public:
31
32         // types
33
34         typedef std::random_access_iterator_tag iterator_category;
35         typedef typename R::value_type value_type;
36         typedef std::size_t size_type;
37         typedef std::ptrdiff_t difference_type;
38         typedef typename if_then_else<is_const, value_type const*, value_type*>::type
39             pointer;
40         typedef typename if_then_else<is_const, typename R::const_reference, typename R
41             ::reference>::type reference;
42
43         typedef E entry;
44         typedef typename R::surrogate_type surrogate_type;
45
46     protected:
47
48         // types
49
50         typedef typename if_then_else<is_const, E const*, E*>::type node_pointer;
51
52     public:
53
54         // friends
55
56         friend class rank_iterator<R, !is_const, E>;
57     };
58 }
```

```

56     template <typename V, typename A, typename S, typename I, typename J>
57     friend class cphstl::vector;
58
59     // structs
60
61     rank_iterator();
62     rank_iterator(rank_iterator<R, false, E> const&);
63     rank_iterator(rank_iterator<R, true, E> const&);
64     rank_iterator& operator=(rank_iterator const&);
65     ~rank_iterator();
66
67     // operators
68
69     reference operator*() const;
70     pointer operator->() const;
71     rank_iterator& operator++();
72     rank_iterator operator++(int);
73     rank_iterator& operator--();
74     rank_iterator operator--(int);
75     rank_iterator& operator+=(difference_type);
76     rank_iterator& operator-=(difference_type);
77     rank_iterator operator+(difference_type) const;
78     rank_iterator operator-(difference_type) const;
79     difference_type operator-(rank_iterator const&) const;
80
81     template <bool both>
82     bool operator==(rank_iterator<R, both, E> const&) const;
83
84     template <bool both>
85     bool operator!=(rank_iterator<R, both, E> const&) const;
86
87     template <bool both>
88     bool operator<(rank_iterator<R, both, E> const&) const;
89
90     template <bool both>
91     bool operator>(rank_iterator<R, both, E> const&) const;
92
93     template <bool both>
94     bool operator<=(rank_iterator<R, both, E> const&) const;
95
96     template <bool both>
97     bool operator>=(rank_iterator<R, both, E> const&) const;
98
99 protected:
100     // converters to be used by the container friends
101     rank_iterator(std::pair<size_type, surrogate_type*> const&);
102     operator std::pair<size_type, surrogate_type*>() const;
103
104     void advance(difference_type const& n);
105
106     surrogate_type* surrogate;
107     size_type position;
108 };
109
110 template<typename R, bool both, typename E>
111 rank_iterator<R, both, E>
112 operator+(typename R::difference_type, rank_iterator<R, both, E> const&);
113
114 }
115
116 #include "rank-iterator.i++" // implements cphstl::rank_iterator
117
118 #endif

```

7.2 Iterator/Code/rank-iterator.i++

```

1  /*
2   Implementation of cphstl::rank_iterator
3
4   Authors: Jyrki Katajainen, Bo Simonsen © 2008
5  */
6
7  #include <cassert> // defines assert macro
8  #include <iostream>
9
10 namespace cphstl {
11
12     // default constructor
13
14     template <typename R, bool is_const, typename E>
15     rank_iterator<R, is_const, E>::rank_iterator()
16     : surrogate(0), position(size_type()) {
17     }
18
19     // copy constructor
20
21     template <typename R, bool is_const, typename E>
22     rank_iterator<R, is_const, E>::rank_iterator(rank_iterator<R, false, E> const& a)
23     : surrogate(a.surrogate), position(a.position) {
24     }
25
26     template <typename R, bool is_const, typename E>
27     rank_iterator<R, is_const, E>::rank_iterator(rank_iterator<R, true, E> const& a)
28     : surrogate(a.surrogate), position(a.position) {
29     }
30
31     // assignment
32
33     template <typename R, bool is_const, typename E>
34     rank_iterator<R, is_const, E>&
35     rank_iterator<R, is_const, E>::operator=(rank_iterator<R, is_const, E> const& a) {
36         (*this).surrogate = a.surrogate;
37         (*this).position = a.position;
38         return *this;
39     }
40
41     // destructor
42
43     template <typename R, bool is_const, typename E>
44     rank_iterator<R, is_const, E>::~~rank_iterator() {
45     }
46
47     // operator*
48
49     template <typename R, bool is_const, typename E>
50     typename rank_iterator<R, is_const, E>::reference
51     rank_iterator<R, is_const, E>::operator*() const {
52         return reference((*(*(this).surrogate).subject()).access((this).position));
53     }
54
55     // operator->
56
57     template <typename R, bool is_const, typename E>
58     typename rank_iterator<R, is_const, E>::pointer
59     rank_iterator<R, is_const, E>::operator->() const {
60         return pointer(&(*(this).position).content());
61     }
62
63     template <typename R, bool is_const, typename E>

```

```

64 void
65 rank_iterator<R, is_const, E>::advance(difference_type const& n) {
66     if (n == 0)
67         return;
68     (*this).position += n;
69 }
70
71 // operator++; pre-increment
72
73 template <typename R, bool is_const, typename E>
74 rank_iterator<R, is_const, E>&
75 rank_iterator<R, is_const, E>::operator++() {
76     (*this).advance(1);
77     return *this;
78 }
79
80 // operator++; post-increment
81
82 template <typename R, bool is_const, typename E>
83 rank_iterator<R, is_const, E>
84 rank_iterator<R, is_const, E>::operator++(int) {
85     rank_iterator<R, is_const, E> temporary = *this;
86     (*this).advance(1);
87     return temporary;
88 }
89
90 // operator--; pre-decrement
91
92 template <typename R, bool is_const, typename E>
93 rank_iterator<R, is_const, E>&
94 rank_iterator<R, is_const, E>::operator--() {
95     (*this).advance(-1);
96     return *this;
97 }
98
99 // operator--; post-decrement
100
101 template <typename R, bool is_const, typename E>
102 rank_iterator<R, is_const, E>
103 rank_iterator<R, is_const, E>::operator--(int) {
104     rank_iterator<R, is_const, E> temporary = *this;
105     (*this).advance(-1);
106     return temporary;
107 }
108
109 // operator+=
110
111 template <typename R, bool is_const, typename E>
112 rank_iterator<R, is_const, E>&
113 rank_iterator<R, is_const, E>::operator+=(difference_type n) {
114     (*this).advance(n);
115     return *this;
116 }
117
118 // operator-=
119
120 template <typename R, bool is_const, typename E>
121 rank_iterator<R, is_const, E>&
122 rank_iterator<R, is_const, E>::operator-=(difference_type n) {
123     (*this).advance(-n);
124     return *this;
125 }
126
127 // operator+
128

```

```

129 template <typename R, bool is_const, typename E>
130 rank_iterator<R, is_const, E>
131 rank_iterator<R, is_const, E>::operator+(difference_type n) const {
132     rank_iterator<R, is_const, E> temporary = *this;
133     temporary.advance(n);
134     return temporary;
135 }
136
137 // operator-
138
139 template <typename R, bool is_const, typename E>
140 rank_iterator<R, is_const, E>
141 rank_iterator<R, is_const, E>::operator-(difference_type n) const {
142     rank_iterator<R, is_const, E> temporary = *this;
143     temporary.advance(-n);
144     return temporary;
145 }
146
147 // iterator distance
148
149 template <typename R, bool is_const, typename E>
150 typename rank_iterator<R, is_const, E>::difference_type
151 rank_iterator<R, is_const, E>::operator-(rank_iterator<R, is_const, E> const& a)
152     const {
153     return (*this).position - a.position;
154 }
155
156 // operator==
157
158 template <typename R, bool is_const, typename E>
159 template <bool both>
160 bool
161 rank_iterator<R, is_const, E>::operator==(rank_iterator<R, both, E> const& a)
162     const {
163     return (*this).position == a.position;
164 }
165
166 // operator!=
167
168 template <typename R, bool is_const, typename E>
169 template <bool both>
170 bool
171 rank_iterator<R, is_const, E>::operator!=(rank_iterator<R, both, E> const& a)
172     const {
173     return !(*this == a);
174 }
175
176 // operator<
177
178 template <typename R, bool is_const, typename E>
179 template <bool both>
180 bool
181 rank_iterator<R, is_const, E>::operator<(rank_iterator<R, both, E> const& a) const
182     {
183     return ((*this) - a) < 0;
184 }
185
186 // operator>
187
188 template <typename R, bool is_const, typename E>
189 template <bool both>
190 bool
191 rank_iterator<R, is_const, E>::operator>(rank_iterator<R, both, E> const& a) const
192     {
193     return a < *this;

```

```

189 }
190
191 // operator<=
192
193 template <typename R, bool is_const, typename E>
194 template <bool both>
195 bool
196 rank_iterator<R, is_const, E>::operator<=(rank_iterator<R, both, E> const& a)
197     const {
198     return !(a < *this);
199 }
200
201 // operator>=
202
203 template <typename R, bool is_const, typename E>
204 template <bool both>
205 bool
206 rank_iterator<R, is_const, E>::operator>=(rank_iterator<R, both, E> const& a)
207     const {
208     return !(*this < a);
209 }
210
211 // operator+(int, iterator)
212
213 template <typename R, bool is_const, typename E>
214 rank_iterator<R, is_const, E> operator+(typename R::difference_type n,
215     rank_iterator<R, is_const, E> const& a) {
216     return a + n;
217 }
218
219 // parametrized constructor
220
221 template <typename R, bool is_const, typename E>
222 rank_iterator<R, is_const, E>::rank_iterator(std::pair<size_type, surrogate_type*>
223     const& p)
224     : surrogate(p.second), position(p.first) {
225 }
226
227 // conversion operator
228
229 template <typename R, bool is_const, typename E>
230 rank_iterator<R, is_const, E>::operator std::pair<size_type, surrogate_type*>()
231     const {
232     return std::pair<size_type, surrogate_type*>((*this).position, (*this).surrogate
233     );
234 }

```

7.3 Iterator/Code/proxy-iterator.h++

```

1 /*
2  The proxy iterator stores a pointer to a encapsulator and a pointer
3  to a surrogate. The pointer to the encapsulator represents the current
4  position of where the value can be fetched. The surrogate points at
5  the kernel. The surrogate is used to advanced the iterator, where the
6  access member function in the kernel is called to get a pointer to
7  the desired encapsulator.
8
9  The idea of combining iterators and const iterators into the same
10 class is taken from [Matt Austern. Defining iterators and const
11 iterators. C/C++ User's Journal 19,1 (2001), 74-79].
12
13 Author: Jyrki Katajainen, Bo Simonsen, April 2008

```

```

14 */
15
16 #ifndef __CPHSTL_SAFE_ENTRY_ITERATOR__
17 #define __CPHSTL_SAFE_ENTRY_ITERATOR__
18
19 #include <iterator> // defines std::random_access_iterator_tag
20 #include <cstddef> // defines std::size_t and std::ptrdiff_t
21 #include <utility> // defines std::pair
22
23 #include "type.h++" // defines cphstl::if_then_else
24
25 namespace cphstl {
26
27     template <typename V, typename A, typename R, typename I, typename J>
28     class vector;
29
30     template <typename R, bool is_const = false, typename E = typename R::
31         encapsulator_type>
32     class proxy_iterator {
33     public:
34
35         // types
36
37         typedef std::random_access_iterator_tag iterator_category;
38         typedef typename R::value_type value_type;
39         typedef std::size_t size_type;
40         typedef std::ptrdiff_t difference_type;
41         typedef typename if_then_else<is_const, value_type const*, value_type*>::type
42             pointer;
43         typedef typename if_then_else<is_const, typename R::const_reference, typename R
44             ::reference>::type reference;
45
46         typedef E encapsulator_type;
47         typedef typename R::surrogate_type surrogate_type;
48
49     protected:
50
51         // types
52
53         typedef typename if_then_else<is_const, E const*, E*>::type node_pointer;
54
55     public:
56
57         // friends
58
59         friend class proxy_iterator<R, !is_const, E>;
60
61         template <typename V, typename A, typename S, typename I, typename J>
62         friend class cphstl::vector;
63
64         // structs
65
66         proxy_iterator();
67         proxy_iterator(proxy_iterator<R, false, E> const&);
68         proxy_iterator(proxy_iterator<R, true, E> const&);
69         proxy_iterator& operator=(proxy_iterator const&);
70         ~proxy_iterator();
71
72         // operators
73
74         reference operator*() const;
75         pointer operator->() const;
76         proxy_iterator& operator++();
77         proxy_iterator operator++(int);

```

```

76     proxy_iterator& operator--();
77     proxy_iterator operator--(int);
78     proxy_iterator& operator+=(difference_type);
79     proxy_iterator& operator-=(difference_type);
80     proxy_iterator operator+(difference_type) const;
81     proxy_iterator operator-(difference_type) const;
82     difference_type operator-(proxy_iterator const&) const;
83
84     template <bool both>
85     bool operator==(proxy_iterator<R, both, E> const&) const;
86
87     template <bool both>
88     bool operator!=(proxy_iterator<R, both, E> const&) const;
89
90     template <bool both>
91     bool operator<(proxy_iterator<R, both, E> const&) const;
92
93     template <bool both>
94     bool operator>(proxy_iterator<R, both, E> const&) const;
95
96     template <bool both>
97     bool operator<=(proxy_iterator<R, both, E> const&) const;
98
99     template <bool both>
100    bool operator>=(proxy_iterator<R, both, E> const&) const;
101
102    protected:
103        // converters to be used by the container friends
104        proxy_iterator(std::pair<size_type, surrogate_type*> const&);
105        operator std::pair<size_type, surrogate_type*>() const;
106
107        void advance(difference_type n);
108
109        surrogate_type* surrogate;
110        node_pointer position;
111    };
112
113    template<typename R, bool both, typename E>
114    proxy_iterator<R, both, E>
115    operator+(typename R::difference_type, proxy_iterator<R, both, E> const&);
116
117 }
118
119 #include "proxy-iterator.i++" // implements cphstl::proxy_iterator
120
121 #endif

```

7.4 Iterator/Code/proxy-iterator.i++

```

1 /*
2  * Implementation of cphstl::proxy_iterator
3  *
4  * Authors: Jyrki Katajainen, Bo Simonsen © 2008
5  */
6
7 #include <cassert> // defines assert macro
8 #include <iostream>
9
10 namespace cphstl {
11
12     // default constructor
13
14     template <typename R, bool is_const, typename E>
15     proxy_iterator<R, is_const, E>::proxy_iterator()
16     : surrogate(0), position(node_pointer()) {

```

```

17 }
18
19 // copy constructor
20
21 template <typename R, bool is_const, typename E>
22 proxy_iterator<R, is_const, E>::proxy_iterator(proxy_iterator<R, false, E> const&
23     a)
24     : surrogate(a.surrogate), position(a.position) {
25 }
26
27 template <typename R, bool is_const, typename E>
28 proxy_iterator<R, is_const, E>::proxy_iterator(proxy_iterator<R, true, E> const&
29     a)
30     : surrogate(a.surrogate), position(a.position) {
31 }
32
33 // assignment
34
35 template <typename R, bool is_const, typename E>
36 proxy_iterator<R, is_const, E>&
37 proxy_iterator<R, is_const, E>::operator=(proxy_iterator<R, is_const, E> const& a)
38 {
39     (*this).surrogate = a.surrogate;
40     (*this).position = a.position;
41     return *this;
42 }
43
44 // destructor
45
46 template <typename R, bool is_const, typename E>
47 proxy_iterator<R, is_const, E>::~~proxy_iterator() {
48 }
49
50 // operator*
51
52 template <typename R, bool is_const, typename E>
53 typename proxy_iterator<R, is_const, E>::reference
54 proxy_iterator<R, is_const, E>::operator*() const {
55     return reference((*this).position);
56 }
57
58 // operator->
59
60 template <typename R, bool is_const, typename E>
61 typename proxy_iterator<R, is_const, E>::pointer
62 proxy_iterator<R, is_const, E>::operator->() const {
63     return pointer(&((*this).position).content());
64 }
65
66 template <typename R, bool is_const, typename E>
67 void
68 proxy_iterator<R, is_const, E>::advance(difference_type n) {
69     if (n == 0)
70         return;
71
72     if ((*this).position == 0) {
73         (*this).position = ((*surrogate).subject()).access((*(*this).surrogate).
74             subject()).size() + n);
75     }
76     else {
77         if ((*(*this).position).position() + n >= ((*(*this).surrogate).subject()).
78             size()) {
79             (*this).position = 0;
80         }
81         else {

```

```

77         (*this).position = ((*surrogate).subject()).access((*this).position).
           position() + n);
78     }
79 }
80 }
81
82 // operator++; pre-increment
83
84 template <typename R, bool is_const, typename E>
85 proxy_iterator<R, is_const, E>&
86 proxy_iterator<R, is_const, E>::operator++() {
87     (*this).advance(1);
88     return *this;
89 }
90
91 // operator++; post-increment
92
93 template <typename R, bool is_const, typename E>
94 proxy_iterator<R, is_const, E>
95 proxy_iterator<R, is_const, E>::operator++(int) {
96     proxy_iterator<R, is_const, E> temporary = *this;
97     (*this).advance(1);
98     return temporary;
99 }
100
101 // operator--; pre-decrement
102
103 template <typename R, bool is_const, typename E>
104 proxy_iterator<R, is_const, E>&
105 proxy_iterator<R, is_const, E>::operator--() {
106     (*this).advance(-1);
107     return *this;
108 }
109
110 // operator--; post-decrement
111
112 template <typename R, bool is_const, typename E>
113 proxy_iterator<R, is_const, E>
114 proxy_iterator<R, is_const, E>::operator--(int) {
115     proxy_iterator<R, is_const, E> temporary = *this;
116     (*this).advance(-1);
117     return temporary;
118 }
119
120 // operator+=
121
122 template <typename R, bool is_const, typename E>
123 proxy_iterator<R, is_const, E>&
124 proxy_iterator<R, is_const, E>::operator+=(difference_type n) {
125     (*this).advance(n);
126     return *this;
127 }
128
129 // operator-=
130
131 template <typename R, bool is_const, typename E>
132 proxy_iterator<R, is_const, E>&
133 proxy_iterator<R, is_const, E>::operator-=(difference_type n) {
134     (*this).advance(-n);
135     return *this;
136 }
137
138 // operator+
139
140 template <typename R, bool is_const, typename E>

```

```

141 proxy_iterator<R, is_const, E>
142 proxy_iterator<R, is_const, E>::operator+(difference_type n) const {
143     proxy_iterator<R, is_const, E> temporary = *this;
144     temporary.advance(n);
145     return temporary;
146 }
147
148 // operator-
149
150 template <typename R, bool is_const, typename E>
151 proxy_iterator<R, is_const, E>
152 proxy_iterator<R, is_const, E>::operator-(difference_type n) const {
153     proxy_iterator<R, is_const, E> temporary = *this;
154     temporary.advance(-n);
155     return temporary;
156 }
157
158 // iterator distance
159
160 template <typename R, bool is_const, typename E>
161 typename proxy_iterator<R, is_const, E>::difference_type
162 proxy_iterator<R, is_const, E>::operator-(proxy_iterator<R, is_const, E> const& a)
163     const {
164     if ((*this).position == 0 && a.position == 0) {
165         return 0;
166     }
167     else if ((*this).position == 0) {
168         return ((*this).surrogate().subject()).size() - (*a.position).position();
169     }
170     else if (a.position == 0) {
171         return ((*this).position).position() - ((*a.position).surrogate().subject()).size();
172     }
173     else {
174         return ((*this).position).position() - (*a.position).position();
175     }
176 }
177 // operator==
178
179 template <typename R, bool is_const, typename E>
180 template <bool both>
181 bool
182 proxy_iterator<R, is_const, E>::operator==(proxy_iterator<R, both, E> const& a)
183     const {
184     return (*this).surrogate == a.surrogate && (*this).position == a.position;
185 }
186 // operator!=
187
188 template <typename R, bool is_const, typename E>
189 template <bool both>
190 bool
191 proxy_iterator<R, is_const, E>::operator!=(proxy_iterator<R, both, E> const& a)
192     const {
193     return !(*this == a);
194 }
195 // operator<
196
197 template <typename R, bool is_const, typename E>
198 template <bool both>
199 bool
200 proxy_iterator<R, is_const, E>::operator<(proxy_iterator<R, both, E> const& a)
201     const {

```

```

201     return ((*this) - a) < 0;
202 }
203
204 // operator>
205
206 template <typename R, bool is_const, typename E>
207 template <bool both>
208 bool
209 proxy_iterator<R, is_const, E>::operator>(proxy_iterator<R, both, E> const& a)
210     const {
211     return a < *this;
212 }
213
214 // operator<=
215
216 template <typename R, bool is_const, typename E>
217 template <bool both>
218 bool
219 proxy_iterator<R, is_const, E>::operator<=(proxy_iterator<R, both, E> const& a)
220     const {
221     return !(a < *this);
222 }
223
224 // operator>=
225
226 template <typename R, bool is_const, typename E>
227 template <bool both>
228 bool
229 proxy_iterator<R, is_const, E>::operator>=(proxy_iterator<R, both, E> const& a)
230     const {
231     return !(*this < a);
232 }
233
234 // operator+(int, iterator)
235
236 template <typename R, bool is_const, typename E>
237 proxy_iterator<R, is_const, E> operator+(typename R::difference_type n,
238     proxy_iterator<R, is_const, E> const& a) {
239     return a + n;
240 }
241
242 // parametrized constructor
243
244 template <typename R, bool is_const, typename E>
245 proxy_iterator<R, is_const, E>::proxy_iterator(std::pair<size_type, surrogate_type
246     *> const& p)
247     : surrogate(p.second) {
248     if ((*surrogate).subject().size() == p.first) {
249         position = 0;
250     }
251     else {
252         position = ((*surrogate).subject()).access(p.first);
253     }
254 }
255
256 // conversion operator
257
258 template <typename R, bool is_const, typename E>
259 proxy_iterator<R, is_const, E>::operator std::pair<size_type, surrogate_type*>()
260     const {
261     if (position == 0) {
262         return std::pair<size_type, surrogate_type*>((*surrogate).subject().size(),
263             (*this).surrogate);
264     }
265     return std::pair<size_type, surrogate_type*>((*this).position).position(), (*

```

```

        this).surrogate);
259 }
260
261 }

```

8. Kernels

8.1 Vector-framework/Code/dynamic-array.h++

```

1 #ifndef __CPHSTL_DYNAMIC_ARRAY__
2 #define __CPHSTL_DYNAMIC_ARRAY__
3
4 /*
5  The dynamic array kernel.
6  Expansion factor 2, Contraction threshold 1/4.
7  If template argument fast is set, contraction threshold 0 will be used.
8
9  Authors: Jyrki Katajainen, Bo Simonsen, May 2008
10 */
11
12 #include <cstdlib> // defines std::size_t
13 #include "direct-encapsulator.h++"
14 #include "type.h++"
15 #include "uninitialized-copy.i++"
16 #include "allocator-proxy.h++"
17
18 namespace cphstl {
19
20     template <typename V, typename A, typename K>
21     class vector_framework;
22
23     template <
24         typename V,
25         typename A,
26         typename E = direct_encapsulator<V, A>,
27         bool fast = false
28     >
29     class dynamic_array {
30
31     public:
32         typedef std::size_t size_type;
33         typedef E encapsulator_type;
34         typedef typename E::segment_type segment_type;
35         typedef typename E::slot_type slot_type;
36         typedef typename A::template rebind<slot_type>::other segment_allocator_type;
37
38         dynamic_array(A const& a = A())
39             : segment_allocator(a), segment(0), current_capacity(0), current_size(0) {
40         }
41
42         ~dynamic_array() {
43         }
44
45         size_type max_size() const {
46             typename A::template rebind<char>::other byte_allocator((*this).
47                 segment_allocator.subject());
48             return byte_allocator.max_size() / (5 * sizeof(slot_type) + E::size);
49         }
50
51         size_type size() const {
52             return (*this).current_size;
53         }
54
55         void size(size_type s) {

```

```

55     (*this).current_size = s;
56 }
57
58 size_type capacity() const {
59     return (*this).current_capacity;
60 }
61
62 slot_type& access(size_type index) {
63     return (*this).segment[index];
64 }
65
66 void grow(size_type delta) {
67     size_type new_capacity = (*this).current_capacity == 0 ? 1 : (*this).
        current_capacity;
68     segment_type new_segment = 0;
69
70     do {
71         new_capacity <<= 1;
72     } while (new_capacity < (*this).current_size + delta);
73
74     if (new_capacity >= (*this).max_size()) {
75         new_capacity = (*this).current_size + delta;
76     }
77
78     new_segment = (*this).segment_allocator.allocate(new_capacity);
79
80     cphstl::uninitialized_copy((*this).segment, (*this).segment + (*this).
        current_size, new_segment);
81
82     if ((*this).segment != 0) {
83         (*this).segment_allocator.deallocate((*this).segment, (*this).
            current_capacity);
84     }
85
86     (*this).segment = new_segment;
87     (*this).current_capacity = new_capacity;
88 }
89
90 void shrink() {
91     if(fast) {
92         return;
93     }
94
95     size_type new_capacity = (*this).current_capacity;
96
97     while (new_capacity > (*this).current_size){
98         new_capacity >>= 2;
99     }
100    new_capacity <<= 2;
101
102    if ((*this).current_capacity == new_capacity) {
103        return;
104    }
105
106    segment_type new_segment = 0;
107
108    if(new_capacity != 0) {
109        new_segment = segment_allocator.allocate(new_capacity);
110        cphstl::uninitialized_copy((*this).segment, (*this).segment + (*this).
            current_size, new_segment);
111    }
112
113    if ((*this).segment != 0) {
114        segment_allocator.deallocate((*this).segment, (*this).current_capacity);
115    }

```

```

116
117     (*this).segment = new_segment;
118     (*this).current_capacity = new_capacity;
119 }
120
121 private:
122     allocator_proxy<segment_allocator_type> segment_allocator;
123     segment_type segment;
124     size_type current_capacity;
125     size_type current_size;
126 };
127 }
128
129 #endif

```

8.2 Vector-framework/Code/hashed-array-tree.h++

```

1 #ifndef __CPHSTL_HASHED_ARRAY_TREE__
2 #define __CPHSTL_HASHED_ARRAY_TREE__
3
4 /*
5  The hashed array tree kernel
6
7  Authors: Jyrki Katajainen, Bo Simonsen © 2009
8  */
9
10 #include <cmath>
11 #include <algorithm>
12
13 #include "allocator-proxy.h++"
14
15 namespace cphstl {
16     template <typename V, typename A, typename K>
17     class vector_framework;
18     template <
19         typename V,
20         typename A,
21         typename E = direct_encapsulator<V, A>
22     >
23     class hashed_array_tree {
24
25     public:
26         typedef std::size_t size_type;
27         typedef E encapsulator_type;
28         typedef typename E::segment_type segment_type;
29         typedef typename E::slot_type slot_type;
30
31     private:
32         typedef segment_type* directory_type;
33         typedef typename A::template rebind<slot_type>::other segment_allocator_type;
34         typedef typename A::template rebind<segment_type>::other
35             directory_allocator_type;
36
37     public:
38         hashed_array_tree(A const& a = A())
39             : directory_allocator(a), segment_allocator(a), directory(0),
40               current_size(0), segment_size(0), number_of_segments(0), mask(0),
41               power(0) {
42
43         ~hashed_array_tree() {
44
45         size_type max_size() const {
46             typename A::template rebind<char>::other byte_allocator((*this).

```

```

        segment_allocator.subject();
48     float b = byte_allocator.max_size();
49     float e = E::size;
50     float p = sizeof(int*);
51     float m = std::max(e, p);
52     return size_type(floor(sqrt((- 2 * m - 4 + sqrt((m * 2 + 4) * (m * 2 + 4) + 8
        * e * b)) / 4 * e)));
53 }
54
55 size_type size() const {
56     return (*this).current_size;
57 }
58
59 void size(size_type s) {
60     (*this).current_size = s;
61 }
62
63 size_type capacity() const {
64     return (*this).segment_size * (*this).number_of_segments;
65 }
66
67 slot_type& access(size_type index) {
68     return (*this).directory[index >> power][index & mask];
69 }
70
71 void grow(size_type delta) {
72     if ((*this).current_size + delta > (*this).segment_size * (*this).segment_size
73         ) {
74         (*this).reorganize((*this).current_size + delta);
75     }
76     else {
77         size_type new_number_of_segments = (*this).number_of_segments;
78         try {
79             while ((*this).get_number_of_segments((*this).current_size + delta, (*this)
80                 .power, (*this).mask) > new_number_of_segments) {
81                 (*this).directory[new_number_of_segments] = segment_allocator.allocate
82                     ((*this).segment_size);
83                 new_number_of_segments++;
84             }
85         }
86         catch (...) {
87             while (new_number_of_segments > (*this).number_of_segments) {
88                 --new_number_of_segments;
89                 segment_allocator.deallocate((*this).directory[new_number_of_segments],
90                     (*this).segment_size);
91             }
92         }
93     }
94     (*this).number_of_segments = new_number_of_segments;
95 }
96
97 void shrink() {
98     if (8 * (*this).size() < (*this).capacity()) {
99         (*this).reorganize((*this).current_size);
100     }
101     else {
102         while ((*this).get_number_of_segments((*this).current_size, (*this).power,
103             (*this).mask) < (*this).number_of_segments) {
104             --(*this).number_of_segments;
105             segment_allocator.deallocate((*this).directory[(*this).number_of_segments
106                 ], (*this).segment_size);
107         }
108     }
109
110     if((*this).number_of_segments == 0) {

```

```

105         (*this).directory_allocator.deallocate((*this).directory, (*this).
            segment_size);
106     (*this).segment_size = 0;
107     }
108 }
109 }
110
111 private:
112 size_type get_number_of_segments(size_type required_capacity, size_type power,
            size_type mask) {
113     return (required_capacity >> power) + ((required_capacity & mask) > 0 ? 1 : 0)
            ;
114 }
115
116 void reorganize(size_type required_capacity) {
117
118     size_type new_segment_size = 1;
119     size_type new_power = 0;
120
121     while (new_segment_size * new_segment_size < required_capacity) {
122         new_segment_size += new_segment_size;
123         new_power += 1;
124     }
125
126     directory_type new_directory = 0;
127     size_type new_mask = (1 << new_power) - 1;
128     size_type new_number_of_segments = (*this).get_number_of_segments(
            required_capacity, new_power, new_mask);
129
130     if(new_number_of_segments > 0)
131         new_directory = directory_allocator.allocate(new_segment_size);
132
133     size_type i = 0;
134
135     try {
136         while (i < new_number_of_segments) {
137             new_directory[i] = segment_allocator.allocate(new_segment_size);
138             ++i;
139         }
140     }
141     catch (...) {
142         while (i >= 0) {
143             segment_allocator.deallocate(new_directory[i], new_segment_size);
144             --i;
145         }
146         throw;
147     }
148
149     for (size_type i = 0; i < (*this).current_size; ++i) {
150         new (&new_directory[i >> new_power][i & new_mask]) slot_type(directory[i >>
            power][i & mask]);
151     }
152
153     for (size_type i = 0; i < number_of_segments; ++i) {
154         segment_allocator.deallocate(directory[i], segment_size);
155     }
156
157     directory_allocator.deallocate(directory, segment_size);
158
159     (*this).segment_size = new_segment_size;
160     (*this).power = new_power;
161     (*this).mask = new_mask;
162     (*this).number_of_segments = new_number_of_segments;
163     (*this).directory = new_directory;
164 }

```

```

165
166     allocator_proxy<directory_allocator_type> directory_allocator;
167     allocator_proxy<segment_allocator_type> segment_allocator;
168     directory_type directory;
169     size_type current_size;
170     size_type segment_size;
171     size_type number_of_segments;
172     size_type mask;
173     size_type power;
174 };
175 }
176
177 #endif

```

8.3 Vector-framework/Code/levelwise-allocated-pile.hpp

```

1 #ifndef __CPHSTL_LEVELWISE_ALLOCATED_PILE__
2 #define __CPHSTL_LEVELWISE_ALLOCATED_PILE__
3
4 /*
5  The levelwise-allocated pile kernel. Notice, that the directory
6  is of a fixed size.
7
8  Authors: Filip Bruman, Jyrki Katajainen, Mads D. Kristensen, Bjarke
9  Buur Mortensen, Daniel P. Larsen, Bo Simonsen, Christian Wolfgang ©
10 2001, 2004, 2008, 2009
11 */
12
13 #include <cmath> // defines ilogb
14
15 namespace cphstl {
16
17     template <
18         typename V,
19         typename A,
20         typename E = direct_encapsulator<V, A>
21     >
22     class levelwise_allocated_pile {
23
24     public:
25         typedef std::size_t size_type;
26         typedef E encapsulator_type;
27         typedef typename E::segment_type segment_type;
28         typedef typename E::slot_type slot_type;
29         typedef segment_type* directory_type;
30         typedef typename A::template rebind<slot_type>::other segment_allocator_type;
31
32         levelwise_allocated_pile(A const& a = A()) : segment_allocator(a), current_size
33             (0), current_capacity(0), number_of_segments(0), last_segment_size(0) {
34         }
35
36         ~levelwise_allocated_pile() {
37         }
38
39         size_type max_size() const {
40             typename A::template rebind<char>::other byte_allocator((*this).
41                 segment_allocator.subject());
42             size_type max_encapsulators = byte_allocator.max_size() / E::size;
43
44             if (((*this).current_size + max_encapsulators) < max_encapsulators) {
45                 return size_type(-1);
46             }
47
48             return (*this).current_size + max_encapsulators;
49         }
50     };
51 }

```

```

48
49 size_type size() const {
50     return (*this).current_size;
51 }
52
53 void size(size_type s) {
54     (*this).current_size = s;
55 }
56
57 size_type capacity() const {
58     return (*this).current_capacity;
59 }
60
61 slot_type& access(size_type i) {
62     size_type pow = ilogb(i+1);
63     size_type index = i - ((1 << pow) - 1);
64     return (*this).directory[pow][index];
65 }
66
67 void grow(size_type delta) {
68     size_type new_capacity = (*this).current_capacity;
69     size_type new_number_of_segments = (*this).number_of_segments;
70
71     try {
72         while ((*this).current_size + delta > new_capacity) {
73             if ( new_capacity > (*this).max_size() ) {
74                 (*this).directory[new_number_of_segments] = segment_allocator.allocate
75                     ((*this).current_size + delta);
76                 (*this).last_segment_size = (*this).current_size + delta;
77                 ++new_number_of_segments;
78                 new_capacity += delta;
79                 break;
80             } else {
81                 new_capacity += 1 << new_number_of_segments;
82                 (*this).directory[new_number_of_segments] = segment_allocator.allocate(1
83                     << new_number_of_segments);
84                 ++new_number_of_segments;
85             }
86         }
87     } catch (...) {
88         while (new_number_of_segments > (*this).number_of_segments) {
89             --new_number_of_segments;
90             segment_allocator.deallocate((*this).directory[new_number_of_segments], 1
91                 << (new_number_of_segments));
92         }
93     }
94     (*this).current_capacity = new_capacity;
95     (*this).number_of_segments = new_number_of_segments;
96 }
97
98 void shrink() {
99     while ( (((*this).current_capacity - (1 << ((*this).number_of_segments - 1)))
100         > (*this).current_size + 8) ||
101         (*this).current_size == 0) && (*this).number_of_segments > 0) {
102         if ((*this).last_segment_size != 0) {
103             --(*this).number_of_segments;
104             (*this).segment_allocator.deallocate((*this).directory[(*this).
105                 number_of_segments], (*this).last_segment_size);
106             (*this).current_capacity -= (*this).last_segment_size;
107             (*this).last_segment_size = 0;
108         }
109     } else {

```

```

108         --(*this).number_of_segments;
109         (*this).segment_allocator.deallocate((*this).directory[(*this).
            number_of_segments], 1 << ((*this).number_of_segments));
110         (*this).current_capacity = ( 1 << ((*this).number_of_segments) );
111     }
112 }
113 }
114
115 private:
116     segment_type directory[32];
117     allocator_proxy<segment_allocator_type> segment_allocator;
118     size_type current_size;
119     size_type current_capacity;
120     size_type number_of_segments;
121     size_type last_segment_size;
122 };
123 }
124
125 #endif

```

9. Copying

9.1 Vector-framework/Code/slot-swap.i++

```

1 #ifndef __CPHSTL_SLOT_SWAP__
2 #define __CPHSTL_SLOT_SWAP__
3
4 /*
5  slot_swap is used for implementing the generic algorithms
6  block_copy_(forward|backward) located in the vector_framework.
7
8  Authors: Jyrki Katajainen, Bo Simonsen © 2009
9 */
10
11 #include <algorithm>
12
13 namespace cphstl {
14     template <typename V, typename A>
15     class direct_encapsulator;
16     template <typename V, typename A>
17     class indirect_encapsulator;
18     template <typename V, typename A>
19     class doubly_indirect_encapsulator;
20
21     template <typename V, typename A>
22     void slot_swap(indirect_encapsulator<V, A>& t, indirect_encapsulator<V, A>&& s) {
23         std::swap((*t).p, (*s).p);
24         std::swap(t, s);
25     }
26
27     template <typename V, typename A>
28     void slot_swap(doubly_indirect_encapsulator<V, A>& t,
29                 doubly_indirect_encapsulator<V, A>&& s) {
30         std::swap((*t).p, (*s).p);
31         std::swap(t, s);
32     }
33
34     template <typename V, typename A>
35     void slot_swap(direct_encapsulator<V, A>& t, direct_encapsulator<V, A>& s) {
36         std::swap(t, s);
37     }
38 }
39 #endif

```

9.2 Algorithm/Code/uninitialized-copy.i++

```

1  /*
2   This file defines the function uninitialized_copy.
3   The normal behaviour of copy is:
4
5     *p = *q for two iterators p and q
6
7   This causes a problem when the iterator p points at a uninitialized
8   element. This function does constructs p using q (copy construction)
9   which solves the problem.
10
11  Author: Jyrki Katajainen © 2001, 2009
12         Bo Simonsen, 2009
13
14  References
15
16  Andrei Alexandrescu, Modern C++ Design: Generic Programming and
17  Design Patterns Applied, Addison-Wesley (2001), see Section 2.10.5.
18
19  John Maddock and Steve Cleary, C++ type traits, Dr. Dobb's Journal
20  25,10 (2000), 38-44.
21  */
22
23  #include <cstdlib> // defines std::size_t
24  #include <cstring> // defines std::memcpy
25  #include <iterator> // defines standard iterator traits
26  #include <tr1/type_traits> // defines standard type traits
27  #include "type.h++" // defines cphstl::int2type
28
29  namespace cphstl {
30
31      namespace {
32
33          enum copy_algorithms {fast, conservative};
34
35          template <typename I, typename O>
36          O uninitialized_copy(I p, I q, O r, cphstl::int2type<fast>) {
37              std::size_t n = q - p;
38              std::memcpy(r, p, n * sizeof(*p));
39              return r + n;
40          }
41
42          template <typename I, typename O>
43          O uninitialized_copy (I p, I q, O r, cphstl::int2type<conservative>) {
44              typedef typename std::iterator_traits<I>::value_type V;
45
46              for (; p != q; ++p, ++r) {
47                  new (&*r) V(*p);
48              }
49              return r;
50          }
51      }
52
53      template <typename I, typename O>
54      O uninitialized_copy(I p, I q, O r) {
55          typedef typename std::iterator_traits<I>::value_type V;
56          typedef typename std::iterator_traits<O>::value_type U;
57          enum { algorithm =
58              std::tr1::is_pointer<I>::value &&
59              std::tr1::is_pointer<O>::value &&
60              std::tr1::is_pod<V>::value &&
61              std::tr1::is_pod<U>::value &&
62              sizeof(V) == sizeof(U) ? fast : conservative
63          };

```

```

64     return uninitialized_copy(p, q, r, cphstl::int2type<algorithm>());
65 }
66 }

```

10. Tests

10.1 Vector-framework/Test/smoke-test.cpp

```

1  /*
2  2 A smoke test for cphstl::vector; this test is taken (with some minor
3  3 modifications) from the book [P. J. Plauger, Alexander A. Stepanov,
4  4 Meng Lee, and David R. Musser, The C++ Standard Template Library,
5  5 Prentice Hall PTR (2001)].
6  6
7  7 Author: Jyrki Katajainen, April 2008
8  8 Modified by Bo Simonsen, November 2008
9  9 */
10
11 #include <memory> // defines std::allocator
12 #include <cstdlib> // defines std::size_t and std::ptrdiff_t
13 #include <cassert> // defines assert
14 #include <vector> // defines std::vector
15 #include <iostream>
16
17 #include "direct-encapsulator.h++"
18 #include "indirect-encapsulator.h++"
19 #include "doubly-indirect-encapsulator.h++"
20
21 #include "dynamic-array.h++"
22 #include "hashed-array-tree.h++"
23 #include "levelwise-allocated-pile.h++"
24
25 #ifndef STD
26     #include "vector-framework.h++"
27 #endif
28
29 #include "proxy-iterator.h++"
30 #include "rank-iterator.h++"
31 #include "stl-vector.h++" // defines cphstl::vector
32
33 template <typename T>
34 inline void ignore_unused_variable_warning(T const&) {
35 }
36
37 template <typename V>
38 void test_vector(V* a, std::size_t n) {
39
40     assert(n > 0);
41     // assert(cphstl::is_sorted(a, a + n));
42     typedef std::allocator<V> A;
43
44     #ifndef STD
45     #define KERNEL std::vector<V, A>
46     #define ITERATOR
47     #elif PLAIN
48     #define KERNEL cphstl::vector_framework< V, A, cphstl::dynamic_array<V, A, cphstl
49         ::direct_encapsulator<V, A>>>
49     #define ITERATOR ,cphstl::rank_iterator< KERNEL, false>, cphstl::rank_iterator<
50         KERNEL, true>
51     #elif SAFE
52     #define KERNEL cphstl::vector_framework< V, A, cphstl::dynamic_array<V, A, cphstl
53         ::indirect_encapsulator<V, A>>>
54     #define ITERATOR ,cphstl::proxy_iterator<KERNEL,false>,cphstl::proxy_iterator<
55         KERNEL, true>

```

```

53 #elif EXTRA_SAFE
54 #define KERNEL cphstl::vector_framework< V, A, cphstl::dynamic_array<V, A, cphstl
    ::doubly_indirect_encapsulator<V, A> > >
55 #define ITERATOR ,cphstl::proxy_iterator<KERNEL,false>,cphstl::proxy_iterator<
    KERNEL, true>
56 #elif SAFE_HAT
57 #define KERNEL cphstl::vector_framework< V, A, cphstl::hashed_array_tree<V, A,
    cphstl::indirect_encapsulator<V, A> > >
58 #define ITERATOR ,cphstl::proxy_iterator<KERNEL,false>,cphstl::proxy_iterator<
    KERNEL, true>
59 #elif HAT
60 #define KERNEL cphstl::vector_framework< V, A, cphstl::hashed_array_tree<V, A,
    cphstl::direct_encapsulator<V, A> > >
61 #define ITERATOR ,cphstl::rank_iterator<KERNEL,false>,cphstl::rank_iterator<KERNEL
    , true>
62 #elif LAP
63 #define KERNEL cphstl::vector_framework< V, A, cphstl::levelwise_allocated_pile<V,
    A, cphstl::direct_encapsulator<V, A> > >
64 #define ITERATOR ,cphstl::rank_iterator<KERNEL,false>,cphstl::rank_iterator<KERNEL
    , true>
65 #elif SAFE_LAP
66 #define KERNEL cphstl::vector_framework< V, A, cphstl::levelwise_allocated_pile<V,
    A, cphstl::doubly_indirect_encapsulator<V, A> > >
67 #define ITERATOR ,cphstl::proxy_iterator<KERNEL,false>,cphstl::proxy_iterator<
    KERNEL, true>
68 #endif
69
70 typedef cphstl::vector<V, A, KERNEL ITERATOR > container;
71
72 V const x = a[0];
73 V y = x;
74 V const z = a[n - 1];
75 typename container::allocator_type* p_alloc = (A*)0;
76 ignore_unused_variable_warning(p_alloc);
77 typename container::pointer p_ptr = (V*)0;
78 ignore_unused_variable_warning(p_ptr);
79 typename container::const_pointer p_cptr = (V const*)0;
80 ignore_unused_variable_warning(p_cptr);
81 typename container::reference p_ref = y;
82 ignore_unused_variable_warning(p_ref);
83 typename container::const_reference p_cref = (V const&) y;
84 ignore_unused_variable_warning(p_cref);
85 typename container::size_type* p_size = (std::size_t*) 0;
86 ignore_unused_variable_warning(p_size);
87 typename container::difference_type* p_diff = (std::ptrdiff_t*) 0;
88 ignore_unused_variable_warning(p_diff);
89 typename container::value_type* p_value = (V*) 0;
90 ignore_unused_variable_warning(p_value);
91
92 container v0;
93 assert(v0.empty());
94 assert(v0.size() == 0);
95 A al = v0.get_allocator();
96 container v0a(al);
97 assert(v0a.size() == 0);
98 assert(v0a.get_allocator() == al);
99 container v1(5);
100 assert(v1.size() == 5);
101 assert(v1.back() == V());
102 container v1a(6, x);
103 assert(v1a.size() == 6);
104 assert(v1a.back() == x);
105 container v1b(7, x, al);
106 assert(v1b.size() == 7);
107 assert(v1b.back() == x);

```

```

108 container v2(v1a);
109
110 assert(v2.size() == 6);
111 assert(v2.front() == x);
112 container v3(v1a.begin(), v1a.end());
113 assert(v3.size() == 6);
114 assert(v3.front() == x);
115 const container v4(v1a.begin(), v1a.end(), a1);
116 assert(v4.size() == 6);
117 assert(v4.front() == x);
118 v0 = v4;
119 assert(v0.size() == 6);
120 assert(v0.front() == x);
121 assert(v0[0] == x);
122 assert(v0.at(5) == x);
123
124 v0.reserve(12);
125 assert(12 <= v0.capacity());
126 v0.resize(8);
127 assert(v0.size() == 8);
128 assert(v0.back() == V());
129 v0.resize(10, z);
130 assert(v0.size() == 10);
131 assert(v0.back() == z);
132 assert(v0.size() <= v0.max_size());
133
134 typedef typename container::iterator iterator;
135 iterator p_it(v0.begin());
136 typedef typename container::const_iterator const_iterator;
137 const_iterator p_cit(v4.begin());
138 typedef typename container::reverse_iterator reverse_iterator;
139 reverse_iterator p_rit(v0.rbegin());
140 typedef typename container::const_reverse_iterator const_reverse_iterator;
141 const_reverse_iterator p_crit(v4.rbegin());
142 assert(*p_it == x);
143 p_it = v0.end();
144 --p_it;
145 assert(*p_it == z);
146 assert(*p_cit == x);
147 p_cit = v4.end();
148 --p_cit;
149 assert(*p_cit == x);
150 assert(*p_rit == z);
151 p_rit = v0.rend();
152 --p_rit;
153 assert(*p_rit == x);
154 assert(*p_crit == x);
155 p_crit = v4.rend();
156 --p_crit;
157 assert(*p_crit == x);
158
159 assert(v0.front() == x);
160 assert(v4.front() == x);
161 v0.push_back(a[0]);
162 assert(v0.back() == a[0]);
163 v0.pop_back();
164 assert(v0.back() == z);
165 assert(v4.back() == x);
166 v0.assign(v4.begin(), v4.end());
167 assert(v0.size() == v4.size());
168 assert(v0.front() == v4.front());
169 v0.assign(4, z);
170 assert(v0.size() == 4);
171 assert(v0.front() == z);
172 p_cit = v0.insert(v0.begin(), x);

```

```

173  assert(*p_cit == x);
174  ++p_cit;
175  assert(v0.front() == x);
176
177  assert(*p_cit == z);
178  v0.insert(v0.begin(), 1, x);
179  p_cit = v0.begin();
180  assert(*p_cit == x);
181  p_cit++;
182  assert(*p_cit == x);
183  ++p_cit;
184  assert(*p_cit == z);
185  v0.insert(v0.end(), v4.begin(), v4.end());
186  assert(v0.back() == v4.back());
187  v0.insert(v0.end(), a, a + n);
188  assert(v0.back() == z);
189  v0.erase(v0.begin());
190  p_cit = v0.begin();
191  assert(*p_cit == x);
192  p_cit++;
193  assert(*p_cit == z);
194  p_it = ++v0.begin();
195  v0.erase(v0.begin(), p_it);
196  assert(v0.front() == z);
197
198  v0.clear();
199  assert(v0.empty());
200  v0.swap(v1);
201  assert(!v0.empty());
202  assert(v1.empty());
203  swap(v0, v1);
204  assert(v0.empty());
205  assert(!v1.empty());
206  assert(v1 == v1);
207  assert(v0 != v1);
208  assert(v0.begin() == v0.end());
209
210  assert(v0 < v1);
211  assert(v1 > v0);
212  assert(v0 <= v1);
213  assert(v1 >= v0);
214 }
215
216 int main(int, char**) {
217     int a[] = {1, 2, 3};
218     test_vector<int>(a, sizeof(a) / sizeof(a[0]));
219
220     char b[] = "abc";
221     test_vector<char>(b, sizeof(b) / sizeof(b[0]) - 1); // \0 removed
222     return 0;
223 }

```

11. UNIX makefile

11.1 Vector-framework/Test/makefile

```

1 CXX = g++
2 CXXFLAGS = -ggdb -pedantic -ansi -x c++ -g -O0
3 PWD = $(shell pwd)
4 CPHSTL_ROOT = $(HOME)/CPHSTL
5 IFLAGS = -I . -I $(CPHSTL_ROOT)/Source/Type/Code -I $(CPHSTL_ROOT)/Source/Iterator/
           Code -I $(CPHSTL_ROOT)/Source/Vector/Code -I $(CPHSTL_ROOT)/Source/Vector-
           framework/Code -I $(CPHSTL_ROOT)/Source/Algorithm/Code/ -I $(CPHSTL_ROOT)/
           Source/Proxy/Code

```

```

6
7 default: smoke_test1 smoke_test2 smoke_test3 smoke_test4 smoke_test5 smoke_test6
      smoke_test7
8
9 x_test:
10     $(CXX) $(CXXFLAGS) $(IFLAGS) x_test.cpp
11     ./a.out
12 #     @rm -f a.out
13
14 smoke_test1:
15     $(CXX) $(CXXFLAGS) -DSAFE $(IFLAGS) smoke-test.cpp -o smoke-test1
16     ./smoke-test1
17     valgrind --leak-check=full ./smoke-test1 2>valgrind.log.1
18     grep "no leaks are possible" valgrind.log.1
19     rm smoke-test1 valgrind.log.1
20 smoke_test2:
21     $(CXX) $(CXXFLAGS) -DEXTRA_SAFE $(IFLAGS) smoke-test.cpp -o smoke-test2
22     ./smoke-test2
23     valgrind --leak-check=full ./smoke-test2 2>valgrind.log.2
24     grep "no leaks are possible" valgrind.log.2
25     rm smoke-test2 valgrind.log.2
26 smoke_test3:
27     $(CXX) $(CXXFLAGS) -DPLAIN $(IFLAGS) smoke-test.cpp -o smoke-test3
28     ./smoke-test3
29     valgrind --leak-check=full ./smoke-test3 2>valgrind.log.3
30     grep "no leaks are possible" valgrind.log.3
31     rm smoke-test3 valgrind.log.3
32 smoke_test4:
33     $(CXX) $(CXXFLAGS) -DSAFE_HAT $(IFLAGS) smoke-test.cpp -o smoke-test4
34     ./smoke-test4
35     valgrind --leak-check=full ./smoke-test4 2>valgrind.log.4
36     grep "no leaks are possible" valgrind.log.4
37     rm smoke-test4 valgrind.log.4
38 smoke_test5:
39     $(CXX) $(CXXFLAGS) -DHAT $(IFLAGS) smoke-test.cpp -o smoke-test5
40     ./smoke-test5
41     valgrind --leak-check=full ./smoke-test5 2>valgrind.log.5
42     grep "no leaks are possible" valgrind.log.5
43     rm smoke-test5 valgrind.log.5
44 smoke_test6:
45     $(CXX) $(CXXFLAGS) -DLAP $(IFLAGS) smoke-test.cpp -o smoke-test6
46     ./smoke-test6
47     valgrind --leak-check=full ./smoke-test6 2>valgrind.log.6
48     grep "no leaks are possible" valgrind.log.6
49     rm smoke-test6 valgrind.log.6
50 smoke_test7:
51     $(CXX) $(CXXFLAGS) -DSAFE_LAP $(IFLAGS) smoke-test.cpp -o smoke-test7
52     ./smoke-test7
53     valgrind --leak-check=full ./smoke-test7 2>valgrind.log.7
54     grep "no leaks are possible" valgrind.log.7
55     rm smoke-test7 valgrind.log.7
56 insert_test: *.cpp *.h++
57     $(CXX) $(CXXFLAGS) -DSP $(IFLAGS) insert_test.cpp -o insert_test_sp
58     $(CXX) $(CXXFLAGS) -DIV $(IFLAGS) insert_test.cpp -o insert_test_iv
59 test: insert_test
60     $(shell ./insert_test_sp > test_sp.txt)
61     $(shell ./insert_test_iv > test_iv.txt)
62     @diff test_sp.txt test_iv.txt > insert_test.txt
63 #     @cat test_sp.txt
64 #     @cat test_iv.txt
65 #     @cat insert_test.txt
66 #     @rm -f test_sp.txt test_iv.txt
67
68 use-test: use-test.cpp
69     $(CXX) $(IFLAGS) -pedantic -ansi -x c++ use-test.cpp

```

```

70     ./a.out
71     rm ./a.out
72
73 example: example.cpp
74     $(CXX) $(CXXFLAGS) $(IFLAGS) example.cpp
75
76 clean:
77     @rm -f *.o *~ a.out smoke_test insert_test_sp insert_test_iv *.txt

```

12. Benchmarks

12.1 Vector-framework/Benchmark/drive.cpp

```

1  #include <iostream> // defines std::cout and std::endl
2  #include <ctime> // defines std::clock
3  #include "experiment.cpp" // defines experiment
4  #include <cstdlib> // defines std::exit()
5  #include <algorithm>
6
7  const std::clock_t reasonable_precision = CLOCKS_PER_SEC * 10;
8  const std::clock_t high_precision = 1 * CLOCKS_PER_SEC;
9
10 double clock_overhead() {
11     std::clock_t k = std::clock();
12     std::clock_t start, limit;
13
14     FILE* fh = fopen("/tmp/clock_overhead", "r");
15     if(fh != 0) {
16         double overhead;
17         if(fread(&overhead, sizeof(double), 1, fh) == 1) {
18             fclose(fh);
19             return overhead;
20         }
21         fclose(fh);
22     }
23
24     // Wait for the clock to tick
25     do
26         start = std::clock();
27     while (start == k);
28
29     // Interrogate the clock until it has advanced
30     limit = start + reasonable_precision;
31
32     unsigned long r = 0;
33     while ((k = std::clock()) < limit)
34         ++r;
35
36     double overhead = double(k - start) / r;
37
38     fh = fopen("/tmp/clock_overhead", "w");
39     fwrite(&overhead, sizeof(double), 1, fh);
40     fclose(fh);
41
42     return overhead;
43 }
44
45 #define MEASURE(running_time, function) { \
46     unsigned long clock_calls = 1; \
47     double overhead = clock_overhead(); \
48     std::clock_t start; \
49     std::clock_t stop; \
50     std::clock_t init = std::clock(); \
51     do \

```

```

52     start = std::clock(); \
53     while (start == init); \
54     function(); \
55     std::clock_t t = std::clock(); \
56     do \
57         ++clock_calls; \
58         while ((stop = std::clock()) == t); \
59         double ticks = double(stop - start) - (double(clock_calls) * overhead); \
60         running_time = ticks / double(CLOCKS_PER_SEC); \
61     }
62
63     /* Define the two modules to use. */
64     #include <functional>
65     #include <vector>
66
67     #ifndef NUMBER_OF_ELEMENTS
68     #define NUMBER_OF_ELEMENTS 100000
69     #endif
70
71     #ifndef FUN_CODE
72     #define FUN_CODE 1
73     #endif
74
75     #define _V unsigned int
76
77     #include "rank-iterator.h++"
78     #include "proxy-iterator.h++"
79     #include "stl-vector.h++"
80     #include "vector-framework.h++"
81     #include "dynamic-array.h++"
82     #include "hashed-array-tree.h++"
83     #include "levelwise-allocated-pile.h++"
84     #include "direct-encapsulator.h++"
85     #include "indirect-encapsulator.h++"
86     #include "doubly-indirect-encapsulator.h++"
87
88     /* ===== 1. Benchmark using std ===== */
89     #if defined(STD)
90     #define A std::allocator<V>
91     #define CONTAINER_TYPE std::vector<V, A>
92     /* ===== 2. Benchmark using cphstl ===== */
93     #elif defined(CPHSTL)
94     #define A std::allocator<V>
95     typedef cphstl::vector_framework<V, A, cphstl::dynamic_array<V, A, cphstl::
        direct_encapsulator<V, A>>> KERNEL;
96     typedef cphstl::rank_iterator< KERNEL, false> ITERATOR;
97     typedef cphstl::rank_iterator< KERNEL, true> CONST_ITERATOR;
98     typedef cphstl::vector<V, A, KERNEL, ITERATOR, CONST_ITERATOR> CONTAINER_TYPE;
99     #elif defined(SAFE_CPHSTL)
100    #define A std::allocator<V>
101    typedef cphstl::vector_framework<V, A, cphstl::dynamic_array<V, A, cphstl::
        indirect_encapsulator<V, A>>> KERNEL;
102    typedef cphstl::proxy_iterator< KERNEL, false> ITERATOR;
103    typedef cphstl::proxy_iterator< KERNEL, true> CONST_ITERATOR;
104    typedef cphstl::vector<V, A, KERNEL, ITERATOR, CONST_ITERATOR> CONTAINER_TYPE;
105    #elif defined(EXTRA_SAFE_CPHSTL)
106    #define A std::allocator<V>
107    typedef cphstl::vector_framework<V, A, cphstl::dynamic_array<V, A, cphstl::
        doubly_indirect_encapsulator<V, A>>> KERNEL;
108    typedef cphstl::proxy_iterator< KERNEL, false> ITERATOR;
109    typedef cphstl::proxy_iterator< KERNEL, true> CONST_ITERATOR;
110    typedef cphstl::vector<V, A, KERNEL, ITERATOR, CONST_ITERATOR> CONTAINER_TYPE;
111    #elif defined(SAFE_HAT_CPHSTL)
112    #define A std::allocator<V>
113    typedef cphstl::vector_framework<V, A, cphstl::hashed_array_tree<V, A, cphstl::

```

```

        indirect_encapsulator<_V, A>>> KERNEL;
114 typedef cphstl::proxy_iterator< KERNEL, false> ITERATOR;
115 typedef cphstl::proxy_iterator< KERNEL, true> CONST_ITERATOR;
116 typedef cphstl::vector<_V, A, KERNEL, ITERATOR, CONST_ITERATOR> CONTAINER_TYPE;
117 #elif defined(HAT_CPHSTL)
118 #define A std::allocator<_V>
119 typedef cphstl::vector_framework<_V, A, cphstl::hashed_array_tree<_V, A, cphstl::
        direct_encapsulator<_V, A>>> KERNEL;
120 typedef cphstl::rank_iterator< KERNEL, false> ITERATOR;
121 typedef cphstl::rank_iterator< KERNEL, true> CONST_ITERATOR;
122 typedef cphstl::vector<_V, A, KERNEL, ITERATOR, CONST_ITERATOR> CONTAINER_TYPE;
123 #elif defined(LAP_CPHSTL)
124 #define A std::allocator<_V>
125 typedef cphstl::vector_framework<_V, A, cphstl::levelwise_allocated_pile<_V, A,
        cphstl::direct_encapsulator<_V, A>>> KERNEL;
126 typedef cphstl::rank_iterator< KERNEL, false> ITERATOR;
127 typedef cphstl::rank_iterator< KERNEL, true> CONST_ITERATOR;
128 typedef cphstl::vector<_V, A, KERNEL, ITERATOR, CONST_ITERATOR> CONTAINER_TYPE;
129 #elif defined(SAFE_LAP_CPHSTL)
130 #define A std::allocator<_V>
131 typedef cphstl::vector_framework<_V, A, cphstl::levelwise_allocated_pile<_V, A,
        cphstl::indirect_encapsulator<_V, A>>> KERNEL;
132 typedef cphstl::proxy_iterator< KERNEL, false> ITERATOR;
133 typedef cphstl::proxy_iterator< KERNEL, true> CONST_ITERATOR;
134 typedef cphstl::vector<_V, A, KERNEL, ITERATOR, CONST_ITERATOR> CONTAINER_TYPE;
135 #endif
136
137 #define EXPERIMENTS 10
138
139 int main() {
140     // Create and instance of class experiment.
141     // FUNCODE is defined as a compiler option in plot.py
142     double arr[EXPERIMENTS];
143     int i;
144
145     for(i=0; i < EXPERIMENTS; ++i) {
146         experiment<_V, CONTAINER_TYPE> e(NUMBER_OF_ELEMENTS, FUN_CODE);
147
148         double running_time;
149         MEASURE(running_time, e.primal)
150         arr[i] = running_time;
151     }
152
153     double running_time = 0;
154     for(i=0; i < EXPERIMENTS; ++i) {
155         running_time += arr[i];
156     }
157
158     // benz uses std::cout so make sure NOT to insert any text in there.
159     std::cout << (1.0e9 * running_time/EXPERIMENTS) << std::endl;
160     return 0;
161 }

```

12.2 Vector-framework/Benchmark/experiment.c++

```

1 #include <vector> // defines std::vector
2 #include <functional> // defines std::less
3 #include <cassert> // defines assert macro
4 #include <algorithm> // defines std::equal
5 #include <iostream> // defines std streams
6
7
8 template <typename V, typename I>
9 void decreasing_sequence(I p, I q) {
10     int i = q - p;

```

```

11 while (p != q) {
12     *p = V(i);
13     --i; // decreasing
14     ++p;
15 }
16 }
17
18 template <typename V, typename I>
19 void increasing_sequence(I p, I q) {
20     int j = 0;
21     while (p != q) {
22         *p = V(j);
23         j++; // increasing
24         ++p;
25     }
26 }
27
28 template <typename I>
29 void show(I p, I q) {
30     while (p != q) {
31         ++p;
32     }
33 }
34
35
36 /* *****
37 * In this class a container of type S that contains elements of
38 * type V is benchmarked. The vector vec1 initially contains n
39 * copies of V(n). The vector vec2 contains n V(). Usually V(n)
40 * evaluates to n and V() evaluates to 0, but depends on
41 * the constructor in V.
42 * *****/
43 template <typename V, typename S>
44 class experiment {
45 public:
46     experiment(unsigned int n, int h) : container(), n(n) {
47         test = h;
48         vec1 = new V[n];
49         increasing_sequence<V>(vec1, vec1 + n); // init vec1 to contain from
50             0...n.
51         if(h != 0) {
52             container.insert(container.begin(), vec1, vec1 + n);
53         }
54         if(h == 4) {
55             std::random_shuffle(vec1, vec1 + n);
56         }
57     }
58     ~experiment() {
59         delete [] vec1;
60     }
61     void primal() {
62
63         /* Parse input. */
64         switch(test) {
65             /* insert(e) */
66             case 0:
67                 test0();
68                 break;
69             /* find(key) */
70             case 1:
71                 test1();
72                 break;
73             case 2:
74                 test2();

```

```

75         break;
76     case 3:
77         test3();
78         break;
79     case 4:
80         test4();
81         break;
82     case 5:
83         test5();
84         break;
85     default:
86         std::cout << "No such test defined!\n";
87     }
88 }
89
90
91 void test0() {
92     for(V i=0; i < n; ++i) {
93         container.push_back(i);
94     }
95 }
96
97 void test1() {
98     while (container.size() > 0) {
99         container.pop_back();
100    }
101 }
102 void test2() {
103 }
104
105 void test3() {
106     for(unsigned int i=0; i < n; ++i) {
107         container[i] = V(0);
108     }
109 }
110 void test4() {
111     for(unsigned int i=0; i < n; ++i) {
112         container[vec1[i]] = V(0);
113     }
114 }
115 void test5() {
116     for(unsigned int k=0; k < 100; ++k) {
117         container.insert(container.begin() + (n/2), k);
118     }
119 }
120
121
122 private:
123     S container;
124     unsigned long n; // Number of elements
125     int test; // Function to run
126     V* vec1;
127 };

```

12.3 Vector-framework/Benchmark/plot-push-back.py

```

1 #! /usr/bin/env python
2 import benz
3 import os
4
5 # fun1, fun2 are used in the compiler option FUNCODE
6 # The value of FUNCODE decides which function to call
7 # in experiment.cpp (must follow numbering in experiment.cpp).
8 #
9 # 0 : insert(e)

```

```

10 # 1 : find(key)
11 # ...
12
13 fun1=0
14 fun2=0
15 fun3=0
16 fun4=0
17 fun5=0
18 fun6=0
19 fun7=0
20 fun8=0
21
22 # ===== CUSTOMIZE THE BENCHMARKS =====
23 # Note that the compiler options in benz.py is:
24 # Benz settings
25 COMPILER = 'g++'
26 # No path with symlinks allowed.
27 PATH_TO_DRIVER = os.getcwd() + os.sep + 'drive.c++'
28
29 # Common test data
30 ELEM_TYPE = 'int'
31
32 # Configure first test
33 MACRO_1 = 'STD'
34 FUN_CODE_1 = 'FUN.CODE=' + str(fun1)
35
36 # Configure second test
37 MACRO_2 = 'CPHSIL'
38 FUN_CODE_2 = 'FUN.CODE=' + str(fun2)
39
40 # Configure thrid test
41 MACRO_3 = 'SAFE.CPHSIL'
42 FUN_CODE_3 = 'FUN.CODE=' + str(fun3)
43
44 # Configure thrid test
45 MACRO_4 = 'EXTRA.SAFE.CPHSIL'
46 FUN_CODE_4 = 'FUN.CODE=' + str(fun4)
47
48 # Configure thrid test
49 MACRO_5 = 'HAT.CPHSIL'
50 FUN_CODE_5 = 'FUN.CODE=' + str(fun6)
51
52 # Configure thrid test
53 MACRO_6 = 'SAFE.HAT.CPHSIL'
54 FUN_CODE_6 = 'FUN.CODE=' + str(fun5)
55
56 # Configure thrid test
57 MACRO_7 = 'LAP.CPHSIL'
58 FUN_CODE_7 = 'FUN.CODE=' + str(fun7)
59
60 # Configure thrid test
61 MACRO_8 = 'SAFE.LAP.CPHSIL'
62 FUN_CODE_8 = 'FUN.CODE=' + str(fun7)
63
64 # ===== CUSTOMIZE THE PLOT =====
65
66 # gnuplot settings
67 TITLE = 'push\\_back_for_integer_data'
68 XLABEL = 'Number_of_operations'
69 YLABEL = 'Execution_time_per_operation_[in_nanoseconds]'
70
71
72 # Configure first test
73 STRING_1 = "std::vector"
74 STRING_2 = "dynamic array, direct containment"

```

```

75 STRING_3 = "dynamic array, indirect containment"
76 STRING_4 = "dynamic array, doubly-indirect containment"
77 STRING_5 = "hashed array tree, direct containment"
78 STRING_6 = "hashed array tree, indirect containment"
79 STRING_7 = "levelwise-allocated pile, direct containment"
80 STRING_8 = "levelwise-allocated pile, indirect containment"
81
82 class case(benz.case):
83     def __init__(self, n, macro, fun):
84         benz.case.__init__(self)
85         self.n = n
86         self.compiler = COMPILER
87         self.compiler_options = ["-I../Code/", "-I../Code/Kernels", "-I../Code/Entries",
88                                 "-I../Code/Factories", "-I../Iterator/Code", "-I../Common", "-I
89                                 ../Type/Code", "-I../Vector/Code", "-I../Algorithm/Code"]
90         self.compiler_options.extend(['-O3', '-DNUMBEROFELEMENTS=' + str(n), '-D' +
91                                     macro, '-D' + fun])
92
93     self.dual_exists = 0
94     self.driver_file = PATH_TO_DRIVER
95
96     def output(self):
97         if self.driver_output != "" and float(self.driver_output) > 0.0:
98             return (self.n, float(self.driver_output) / float(self.n))
99         else:
100             return ()
101
102 # Legend names
103 class curve(benz.curve_suite):
104     def __init__(self, module, macro, fun, log_i, log_k, linetype, pointtype):
105         benz.curve_suite.__init__(self)
106         self.title = module
107         self.linetype = linetype
108         self.pointtype = pointtype
109         for j in range(log_i, log_k):
110             self.add(case(1 << j, macro, fun))
111
112 class plot(benz.plot_suite):
113     def __init__(self, log_i, log_k):
114         benz.plot_suite.__init__(self)
115         self.title = TITLE
116         self.xlabel = XLABEL
117         self.ylabel = YLABEL
118         x = (1 << log_i) - 10
119         y = (1 << log_k) + 10
120         self.gnuplot_commands += 'set key left top Left reverse samples 4 spacing 1.25
121                                 title "\n'
122         self.gnuplot_commands += 'set yrange [1:400]\n'
123         self.gnuplot_commands += 'set xtics ("2^{17}" 131072, "2^{18}" 262144, "2^{19}"
124                                 524288, "2^{20}" 1048576, "2^{21}" 2097152, "2^{22}" 4194304, "2^{23}"
125                                 8388603)\n'
126         self.gnuplot_commands += ""
127     set terminal postscript enhanced \ "Times-Roman" 20
128     set key spacing 1.0
129     set title '%(title)s'
130     set xlabel '%(xlabel)s'
131     set ylabel '%(ylabel)s'
132     set logscale x
133     set pointsize 1.5
134
135 """" % self.__dict__
136     self.add(curve(String_1, Macro_1, Fun_Code_1, log_i, log_k, 1, 6))
137     self.add(curve(String_2, Macro_2, Fun_Code_2, log_i, log_k, 1, 1))
138     self.add(curve(String_3, Macro_3, Fun_Code_3, log_i, log_k, 2, 2))

```

```

134     self.add(curve(String_4, MACRO_4, FUN_CODE_4, log_i, log_k,3,3))
135     self.add(curve(String_5, MACRO_5, FUN_CODE_5, log_i, log_k,1,8))
136     self.add(curve(String_6, MACRO_6, FUN_CODE_6, log_i, log_k,2,10))
137     self.add(curve(String_7, MACRO_7, FUN_CODE_7, log_i, log_k,1,64))
138     self.add(curve(String_8, MACRO_8, FUN_CODE_8, log_i, log_k,2,12))
139
140 if __name__ == '__main__':
141     benz.main(
142         task = plot(17,24),
143         runner = benz.gnuplot_runner
144     )

```

12.4 Vector-framework/Benchmark/plot-pop-back.py

```

1  #!/usr/bin/env python
2  import benz
3  import os
4
5  # fun1, fun2 are used in the compiler option FUNCODE
6  # The value of FUNCODE decides which function to call
7  # in experiment.cpp (must follow numbering in experiment.cpp).
8  #
9  # 0 : insert(e)
10 # 1 : find(key)
11 # ...
12
13 fun1=1
14 fun2=1
15 fun3=1
16 fun4=1
17 fun5=1
18 fun6=1
19 fun7=1
20 fun8=1
21
22 # ===== CUSTOMIZE THE BENCHMARKS =====
23 # Note that the compiler options in benz.py is:
24 # Benz settings
25 COMPILER = 'g++'
26 # No path with symlinks allowed.
27 PATH_TO_DRIVER = os.getcwd() + os.sep + 'drive.c++'
28
29 # Common test data
30 ELEM_TYPE = 'int'
31
32 # Configure first test
33 MACRO_1 = 'SID'
34 FUN_CODE_1 = 'FUN.CODE=' + str(fun1)
35
36 # Configure second test
37 MACRO_2 = 'CPHSIL'
38 FUN_CODE_2 = 'FUN.CODE=' + str(fun2)
39
40 # Configure thrid test
41 MACRO_3 = 'SAFE.CPHSIL'
42 FUN_CODE_3 = 'FUN.CODE=' + str(fun3)
43
44 # Configure thrid test
45 MACRO_4 = 'EXTRA.SAFE.CPHSIL'
46 FUN_CODE_4 = 'FUN.CODE=' + str(fun4)
47
48 # Configure thrid test
49 MACRO_5 = 'HAT.CPHSIL'
50 FUN_CODE_5 = 'FUN.CODE=' + str(fun6)
51

```

```

52 # Configure thrid test
53 MACRO_6 = 'SAFEHAT.CPHSTL'
54 FUN_CODE_6 = 'FUNCODE=' + str(fun5)
55
56 # Configure thrid test
57 MACRO_7 = 'LAP.CPHSTL'
58 FUN_CODE_7 = 'FUNCODE=' + str(fun7)
59
60 # Configure thrid test
61 MACRO_8 = 'SAFELAP.CPHSTL'
62 FUN_CODE_8 = 'FUNCODE=' + str(fun7)
63
64 # ===== CUSTOMIZE THE PLOT =====
65
66 # gnuplot settings
67 TITLE = 'pop\_\_back\_for\_integer\_data'
68 XLABEL = 'Number\_of\_operations'
69 YLABEL = 'Execution\_time\_per\_operation\_ [in\_nanoseconds]'
70
71 # Configure first test
72 STRING_1 = "std::vector"
73 STRING_2 = "dynamic array, direct containment"
74 STRING_3 = "dynamic array, indirect containment"
75 STRING_4 = "dynamic array, doubly-indirect containment"
76 STRING_5 = "hashed array tree, direct containment"
77 STRING_6 = "hashed array tree, indirect containment"
78 STRING_7 = "levelwise-allocated pile, direct containment"
79 STRING_8 = "levelwise-allocated pile, indirect containment"
80
81 class case(benz.case):
82     def __init__(self, n, macro, fun):
83         benz.case.__init__(self)
84         self.n = n
85         self.compiler = COMPILER
86         self.compiler_options = ["-I../Code/", "-I../Code/Kernels", "-I../Code/Entries",
87                                 "-I../Code/Factories", "-I../Iterator/Code", "-I../Common", "-I
88                                 ../Type/Code", "-I../Vector/Code", "-I../Algorithm/Code"]
89         self.compiler_options.extend(['-O3', '-DNUMBEROFELEMENTS=' + str(n), '-D' +
90                                     macro, '-D' + fun])
91
92         self.dual_exists = 0
93         self.driver_file = PATH_TO_DRIVER
94
95     def output(self):
96         if self.driver_output != "" and float(self.driver_output) > 0.0:
97             return (self.n, float(self.driver_output) / float(self.n))
98         else:
99             return ()
100
101 # Legend names
102 class curve(benz.curve_suite):
103     def __init__(self, module, macro, fun, log_i, log_k, linetype, pointtype):
104         benz.curve_suite.__init__(self)
105         self.title = module
106         self.linetype = linetype
107         self.pointtype = pointtype
108         for j in range(log_i, log_k):
109             self.add(case(1 << j, macro, fun))
110
111 class plot(benz.plot_suite):
112     def __init__(self, log_i, log_k):
113         benz.plot_suite.__init__(self)
114         self.title = TITLE
115         self.xlabel = XLABEL

```

```

114     self.ylabel = YLABEL
115     x = (1 << log_i) - 10
116     y = (1 << log_k) + 10
117     self.gnuplot_commands += 'set key left top Left reverse samplen 4 spacing 1.25
    title "\n'
118     self.gnuplot_commands += 'set yrange [1:200]\n'
119     self.gnuplot_commands += 'set xtics ("2^{17}" 131072, "2^{18}" 262144, "2^{19}"
    524288, "2^{20}" 1048576, "2^{21}" 2097152, "2^{22}" 4194304, "2^{23}"
    8388603)\n'
120     self.gnuplot_commands += ""
121 set terminal postscript enhanced \ "Times-Roman" 20
122 set key spacing 1.0
123 set title '%(title)s'
124 set xlabel '%(xlabel)s'
125 set ylabel '%(ylabel)s'
126 set logscale x
127 set pointsize 1.5
128
129 """ % self.__dict__
130     self.add(curve(String_1, MACRO_1, FUN_CODE_1, log_i, log_k, 1, 6))
131     self.add(curve(String_2, MACRO_2, FUN_CODE_2, log_i, log_k, 1, 1))
132     self.add(curve(String_3, MACRO_3, FUN_CODE_3, log_i, log_k, 2, 2))
133     self.add(curve(String_4, MACRO_4, FUN_CODE_4, log_i, log_k, 3, 3))
134     self.add(curve(String_5, MACRO_5, FUN_CODE_5, log_i, log_k, 1, 8))
135     self.add(curve(String_6, MACRO_6, FUN_CODE_6, log_i, log_k, 2, 10))
136     self.add(curve(String_7, MACRO_7, FUN_CODE_7, log_i, log_k, 1, 64))
137     self.add(curve(String_8, MACRO_8, FUN_CODE_8, log_i, log_k, 2, 12))
138
139 if __name__ == '__main__':
140     benz.main(
141         task = plot(17,24),
142         runner = benz.gnuplot_runner
143     )

```

12.5 Vector-framework/Benchmark/plot-sequential-access.py

```

1 #! /usr/bin/env python
2 import benz
3 import os
4
5 # fun1, fun2 are used in the compiler option FUNCODE
6 # The value of FUNCODE decides which function to call
7 # in experiment.cpp (must follow numbering in experiment.cpp).
8 #
9 # 0 : insert(e)
10 # 1 : find(key)
11 # ...
12
13 fun1=3
14 fun2=3
15 fun3=3
16 fun4=3
17 fun5=3
18 fun6=3
19 fun7=3
20 fun8=3
21
22 # ===== CUSTOMIZE THE BENCHMARKS =====
23 # Note that the compiler options in benz.py is:
24 # Benz settings
25 COMPILER = 'g++'
26 # No path with symlinks allowed.
27 PATH_TO_DRIVER = os.getcwd() + os.sep + 'drive.c++'
28
29 # Common test data

```

```

30 ELEM_TYPE      = 'int'
31
32 # Configure first test
33 MACRO_1        = 'STD'
34 FUN_CODE_1     = 'FUNCODE=' + str(fun1)
35
36 # Configure second test
37 MACRO_2        = 'CPHSIL'
38 FUN_CODE_2     = 'FUNCODE=' + str(fun2)
39
40 # Configure third test
41 MACRO_3        = 'SAFE.CPHSIL'
42 FUN_CODE_3     = 'FUNCODE=' + str(fun3)
43
44 # Configure third test
45 MACRO_4        = 'EXTRASAFE.CPHSIL'
46 FUN_CODE_4     = 'FUNCODE=' + str(fun4)
47
48 # Configure third test
49 MACRO_5        = 'HAT.CPHSIL'
50 FUN_CODE_5     = 'FUNCODE=' + str(fun6)
51
52 # Configure third test
53 MACRO_6        = 'SAFE.HAT.CPHSIL'
54 FUN_CODE_6     = 'FUNCODE=' + str(fun5)
55
56 # Configure third test
57 MACRO_7        = 'LAP.CPHSIL'
58 FUN_CODE_7     = 'FUNCODE=' + str(fun7)
59
60 # Configure third test
61 MACRO_8        = 'SAFELAP.CPHSIL'
62 FUN_CODE_8     = 'FUNCODE=' + str(fun7)
63
64 # ===== CUSTOMIZE THE PLOT =====
65
66 # gnuplot settings
67 TITLE          = 'operator [ ]_for_integer_data_(sequential_access)'
68 XLABEL         = 'Number_of_operations'
69 YLABEL         = 'Execution_time_per_operation_[in_nanoseconds]'
70
71 STRING_1       = "std::vector"
72 STRING_2       = "dynamic array, direct containment"
73 STRING_3       = "dynamic array, indirect containment"
74 STRING_4       = "dynamic array, doubly-indirect containment"
75 STRING_5       = "hashed array tree, direct containment"
76 STRING_6       = "hashed array tree, indirect containment"
77 STRING_7       = "levelwise-allocated pile, direct containment"
78 STRING_8       = "levelwise-allocated pile, indirect containment"
79
80 class case(benz.case):
81     def __init__(self, n, macro, fun):
82         benz.case.__init__(self)
83         self.n = n
84         self.compiler = COMPILER
85         self.compiler_options = ["-I../Code/", "-I../Code/Kernels", "-I../Code/Entries",
86                                 "-I../Code/Factories", "-I../Iterator/Code", "-I../Common", "-I
87                                 ../Type/Code", "-I../Vector/Code", "-I../Algorithm/Code"]
88         self.compiler_options.extend(['-O3', '-DNUMBEROFELEMENTS=' + str(n), '-D' +
89                                     macro, '-D' + fun])
89
90     self.dual_exists = 0
91     self.driver_file = PATH_TO_DRIVER
92
93     def output(self):

```

```

92     if self.driver_output != "" and float(self.driver_output) > 0.0:
93         return (self.n, float(self.driver_output) / float(self.n))
94     else:
95         return ()
96
97 # Legend names
98 class curve(benz.curve_suite):
99     def __init__(self, module, macro, fun, log_i, log_k, linetype, pointtype):
100         benz.curve_suite.__init__(self)
101         self.title = module
102         self.linetype = linetype
103         self.pointtype = pointtype
104         for j in range(log_i, log_k):
105             self.add(case(1 << j, macro, fun))
106
107
108 class plot(benz.plot_suite):
109     def __init__(self, log_i, log_k):
110         benz.plot_suite.__init__(self)
111         self.title = TITLE
112         self.xlabel = XLABEL
113         self.ylabel = YLABEL
114         x = (1 << log_i) - 10
115         y = (1 << log_k) + 10
116         self.gnuplot_commands += 'set key left top Left reverse samples 1.25 \
title "\n'
117         self.gnuplot_commands += 'set xtics ("2^{17}" 131072, "2^{18}" 262144, "2^{19}" \
524288, "2^{20}" 1048576, "2^{21}" 2097152, "2^{22}" 4194304, "2^{23}" \
8388603)\n'
118         self.gnuplot_commands += 'set yrange [1:100]\n'
119         self.gnuplot_commands += ""
120 set terminal postscript enhanced \ "Times-Roman" 20
121 set key spacing 1.0
122 set title "%(title)s'
123 set xlabel "%(xlabel)s'
124 set ylabel "%(ylabel)s'
125 set logscale x
126 set pointsize 1.5
127
128 "" % self.__dict__
129     self.add(curve(STRING_1, MACRO_1, FUN_CODE_1, log_i, log_k, 1, 6))
130     self.add(curve(STRING_2, MACRO_2, FUN_CODE_2, log_i, log_k, 1, 1))
131     self.add(curve(STRING_3, MACRO_3, FUN_CODE_3, log_i, log_k, 2, 2))
132     self.add(curve(STRING_4, MACRO_4, FUN_CODE_4, log_i, log_k, 3, 3))
133     self.add(curve(STRING_5, MACRO_5, FUN_CODE_5, log_i, log_k, 1, 8))
134     self.add(curve(STRING_6, MACRO_6, FUN_CODE_6, log_i, log_k, 2, 10))
135     self.add(curve(STRING_7, MACRO_7, FUN_CODE_7, log_i, log_k, 1, 64))
136     self.add(curve(STRING_8, MACRO_8, FUN_CODE_8, log_i, log_k, 2, 12))
137
138 if __name__ == '__main__':
139     benz.main(
140         task = plot(20,25),
141         runner = benz.gnuplot_runner
142     )

```

12.6 Vector-framework/Benchmark/plot-random-access.py

```

1 #! /usr/bin/env python
2 import benz
3 import os
4
5 # fun1, fun2 are used in the compiler option FUNCODE
6 # The value of FUNCODE decides which function to call
7 # in experiment.cpp (must follow numbering in experiment.cpp).
8 #

```

```

9 # 0 : insert(e)
10 # 1 : find(key)
11 # ...
12
13 fun1=4
14 fun2=4
15 fun3=4
16 fun4=4
17 fun5=4
18 fun6=4
19 fun7=4
20 fun8=4
21
22 # ===== CUSTOMIZE THE BENCHMARKS =====
23 # Note that the compiler options in benz.py is:
24 # Benz settings
25 COMPILER = 'g++'
26 # No path with symlinks allowed.
27 PATH_TO_DRIVER = os.getcwd() + os.sep + 'drive.c++'
28
29 # Common test data
30 ELEM_TYPE = 'int'
31
32 # Configure first test
33 MACRO_1 = 'STD'
34 FUN_CODE_1 = 'FUNCODE=' + str(fun1)
35
36 # Configure second test
37 MACRO_2 = 'CPHSIL'
38 FUN_CODE_2 = 'FUNCODE=' + str(fun2)
39
40 # Configure thrid test
41 MACRO_3 = 'SAFE.CPHSIL'
42 FUN_CODE_3 = 'FUNCODE=' + str(fun3)
43
44 # Configure thrid test
45 MACRO_4 = 'EXTRA.SAFE.CPHSIL'
46 FUN_CODE_4 = 'FUNCODE=' + str(fun4)
47
48 # Configure thrid test
49 MACRO_5 = 'HAT.CPHSIL'
50 FUN_CODE_5 = 'FUNCODE=' + str(fun6)
51
52 # Configure thrid test
53 MACRO_6 = 'SAFE.HAT.CPHSIL'
54 FUN_CODE_6 = 'FUNCODE=' + str(fun5)
55
56 # Configure thrid test
57 MACRO_7 = 'LAP.CPHSIL'
58 FUN_CODE_7 = 'FUNCODE=' + str(fun7)
59
60 # Configure thrid test
61 MACRO_8 = 'SAFE.LAP.CPHSIL'
62 FUN_CODE_8 = 'FUNCODE=' + str(fun7)
63
64 # ===== CUSTOMIZE THE PLOT =====
65
66 # gnuplot settings
67 TITLE = 'operator [ ]_for_integer_data_(random_access)'
68 XLABEL = 'Number_of_operations'
69 YLABEL = 'Execution_time_per_operation_[in_nanoseconds]'
70
71
72 STRING_1 = "std::vector"
73 STRING_2 = "dynamic array, direct containment"

```

```

74 STRING_3    = "dynamic array, indirect containment"
75 STRING_4    = "dynamic array, doubly-indirect containment"
76 STRING_5    = "hashed array tree, direct containment"
77 STRING_6    = "hashed array tree, indirect containment"
78 STRING_7    = "levelwise-allocated pile, direct containment"
79 STRING_8    = "levelwise-allocated pile, indirect containment"
80
81 class case(benz.case):
82     def __init__(self, n, macro, fun):
83         benz.case.__init__(self)
84         self.n = n
85         self.compiler = COMPILER
86         self.compiler_options = ["-I../Code/", "-I../Code/Kernels", "-I../Code/Entries",
87                                 "-I../Code/Factories", "-I../Iterator/Code", "-I../Common", "-I
88                                 ../Type/Code", "-I../Vector/Code", "-I../Algorithm/Code"]
89         self.compiler_options.extend(['-O3', '-DNUMBEROFELEMENTS=' + str(n), '-D' +
90                                     macro, '-D' + fun])
91
92     self.dual_exists = 0
93     self.driver_file = PATH_TO_DRIVER
94
95     def output(self):
96         if self.driver_output != "" and float(self.driver_output) > 0.0:
97             return (self.n, float(self.driver_output) / float(self.n))
98         else:
99             return ()
100
101 # Legend names
102 class curve(benz.curve_suite):
103     def __init__(self, module, macro, fun, log_i, log_k, linetype, pointtype):
104         benz.curve_suite.__init__(self)
105         self.title = module
106         self.linetype = linetype
107         self.pointtype = pointtype
108         for j in range(log_i, log_k):
109             self.add(case(1 << j, macro, fun))
110
111 class plot(benz.plot_suite):
112     def __init__(self, log_i, log_k):
113         benz.plot_suite.__init__(self)
114         self.title = TITLE
115         self.xlabel = XLABEL
116         self.ylabel = YLABEL
117         x = (1 << log_i) - 10
118         y = (1 << log_k) + 10
119         self.gnuplot_commands += 'set key left top Left reverse sample 4 spacing 1.25 _
120             title _'\n'
121         self.gnuplot_commands += 'set xtics (_2^{17}' _131072, _2^{18}' _262144, _2^{19}'
122             _524288, _2^{20}' _1048576, _2^{21}' _2097152, _2^{22}' _4194304, _2^{23}' _
123             8388603)\n'
124         self.gnuplot_commands += ""
125     set terminal postscript enhanced \ "Times-Roman" 20
126     set key spacing 1.0
127     set title '%(title)s'
128     set xlabel '%(xlabel)s'
129     set ylabel '%(ylabel)s'
130     set logscale x
131     set logscale y
132     set pointsize 1.5
133
134 """" % self.__dict__
135     self.add(curve(STRING_1, MACRO_1, FUN_CODE_1, log_i, log_k, 1, 6))
136     self.add(curve(STRING_2, MACRO_2, FUN_CODE_2, log_i, log_k, 1, 1))
137     self.add(curve(STRING_3, MACRO_3, FUN_CODE_3, log_i, log_k, 2, 2))

```

```

133     self.add(curve(String_4, Macro_4, Fun_Code_4, log_i, log_k,3,3))
134     self.add(curve(String_5, Macro_5, Fun_Code_5, log_i, log_k,1,8))
135     self.add(curve(String_6, Macro_6, Fun_Code_6, log_i, log_k,2,10))
136     self.add(curve(String_7, Macro_7, Fun_Code_7, log_i, log_k,1,64))
137     self.add(curve(String_8, Macro_8, Fun_Code_8, log_i, log_k,2,12))
138
139     if __name__ == '__main__':
140         benz.main(
141             task = plot(17, 24),
142             runner = benz.gnuplot_runner
143         )

```

12.7 Vector-framework/Benchmark/plot-insert.py

```

1  #!/usr/bin/env python
2  import benz
3  import os
4
5  # fun1, fun2 are used in the compiler option FUNCODE
6  # The value of FUNCODE decides which function to call
7  # in experiment.cpp (must follow numbering in experiment.cpp).
8  #
9  # 0 : insert(e)
10 # 1 : find(key)
11 # ...
12
13 fun1=5
14 fun2=5
15 fun3=5
16 fun4=5
17 fun5=5
18 fun6=5
19 fun7=5
20 fun8=5
21
22 # ===== CUSTOMIZE THE BENCHMARKS =====
23 # Note that the compiler options in benz.py is:
24 # Benz settings
25 COMPILER = 'g++'
26 # No path with symlinks allowed.
27 PATH_TO_DRIVER = os.getcwd() + os.sep + 'drive.c++'
28
29 # Common test data
30 ELEM_TYPE = 'int'
31
32 # Configure first test
33 MACRO_1 = 'STD'
34 FUN_CODE_1 = 'FUNCODE=' + str(fun1)
35
36 # Configure second test
37 MACRO_2 = 'CPHSIL'
38 FUN_CODE_2 = 'FUNCODE=' + str(fun2)
39
40 # Configure thrid test
41 MACRO_3 = 'SAFE.CPHSIL'
42 FUN_CODE_3 = 'FUNCODE=' + str(fun3)
43
44 # Configure thrid test
45 MACRO_4 = 'EXTRA.SAFE.CPHSIL'
46 FUN_CODE_4 = 'FUNCODE=' + str(fun4)
47
48 # Configure thrid test
49 MACRO_5 = 'HAT.CPHSIL'
50 FUN_CODE_5 = 'FUNCODE=' + str(fun6)
51

```

```

52 # Configure thrid test
53 MACRO_6 = 'SAFEHAT.CPHSTL'
54 FUN_CODE_6 = 'FUN.CODE=' + str(fun5)
55
56 # Configure thrid test
57 MACRO_7 = 'LAP.CPHSTL'
58 FUN_CODE_7 = 'FUN.CODE=' + str(fun7)
59
60 # Configure thrid test
61 MACRO_8 = 'SAFELAP.CPHSTL'
62 FUN_CODE_8 = 'FUN.CODE=' + str(fun7)
63
64 # ===== CUSTOMIZE THE PLOT =====
65
66 # gnuplot settings
67 TITLE = 'insert_for_integer_data_(100_insertions_in_the_middle_of_a_
sequence)'
68 XLABEL = 'Number_of_elements'
69 YLABEL = 'Execution_time_per_operation_[in_nanoseconds]'
70
71 STRING_1 = "std::vector"
72 STRING_2 = "dynamic array, direct containment"
73 STRING_3 = "dynamic array, indirect containment"
74 STRING_4 = "dynamic array, doubly-indirect containment"
75 STRING_5 = "hashed array tree, direct containment"
76 STRING_6 = "hashed array tree, indirect containment"
77 STRING_7 = "levelwise-allocated pile, direct containment"
78 STRING_8 = "levelwise-allocated pile, indirect containment"
79
80 class case(benz.case):
81     def __init__(self, n, macro, fun):
82         benz.case.__init__(self)
83         self.n = n
84         self.compiler = COMPILER
85         self.compiler_options = ["-I../Code/", "-I../Code/Kernels", "-I../Code/Entries",
"-I../Code/Factories", "-I../Iterator/Code", "-I../Common", "-I
../Type/Code", "-I../Vector/Code", "-I../Algorithm/Code"]
86         self.compiler_options.extend(['-O3', '-DNUMBEROFELEMENTS=' + str(n), '-D' +
macro, '-D' + fun])
87
88         self.dual_exists = 0
89         self.driver_file = PATH_TO_DRIVER
90
91     def output(self):
92         if self.driver_output != "" and float(self.driver_output) > 0.0:
93             return (self.n, float(self.driver_output) / 100.0)
94         else:
95             return ()
96
97 # Legend names
98 class curve(benz.curve_suite):
99     def __init__(self, module, macro, fun, log_i, log_k, linetype, pointtype):
100         benz.curve_suite.__init__(self)
101         self.title = module
102         self.linetype = linetype
103         self.pointtype = pointtype
104         for j in range(log_i, log_k):
105             self.add(case(1 << j, macro, fun))
106
107
108 class plot(benz.plot_suite):
109     def __init__(self, log_i, log_k):
110         benz.plot_suite.__init__(self)
111         self.title = TITLE
112         self.xlabel = XLABEL

```

```

113     self.ylabel = YLABEL
114     x = (1 << log_i) - 10
115     y = (1 << log_k) + 10
116     self.gnuplot_commands += 'set key left top Left reverse samples 4 spacing 1.25
title_\n'
117     self.gnuplot_commands += 'set xtics ("2^{17}" 131072, "2^{18}" 262144, "2^{19}"
524288, "2^{20}" 1048576, "2^{21}" 2097152, "2^{22}" 4194304, "2^{23}"
8388603)\n'
118     self.gnuplot_commands += ""
119 set terminal postscript enhanced \ "Times-Roman\ " 20
120 set key spacing 1.0
121 set title '%(title)s'
122 set xlabel '%(xlabel)s'
123 set ylabel '%(ylabel)s'
124 set logscale x
125 set logscale y
126 set pointsize 1.5
127
128 """ % self.__dict__
129     self.add(curve(String_1, MACRO_1, FUN_CODE_1, log_i, log_k, 1, 6))
130     self.add(curve(String_2, MACRO_2, FUN_CODE_2, log_i, log_k, 1, 1))
131     self.add(curve(String_3, MACRO_3, FUN_CODE_3, log_i, log_k, 2, 2))
132     self.add(curve(String_4, MACRO_4, FUN_CODE_4, log_i, log_k, 3, 3))
133     self.add(curve(String_5, MACRO_5, FUN_CODE_5, log_i, log_k, 1, 8))
134     self.add(curve(String_6, MACRO_6, FUN_CODE_6, log_i, log_k, 2, 10))
135     self.add(curve(String_7, MACRO_7, FUN_CODE_7, log_i, log_k, 1, 64))
136     self.add(curve(String_8, MACRO_8, FUN_CODE_8, log_i, log_k, 2, 12))
137
138 if __name__ == '__main__':
139     benz.main(
140         task = plot(17, 24),
141         runner = benz.gnuplot_runner
142     )

```

12.8 Vector-framework/Benchmark/makefile

```

1 CXX = g++
2 CXXGFLAGS = -ggdb -Wall -x c++ -I../Code/ -I../Iterator/Code -I../Common -I
../Vector/Code -I../Assert -I../Algorithm/Code -I../Type/Code -
DFUN_CODE=3 -DNUMBER_OF_ELEMENTS=1000000
3 #CXXGFLAGS = -Wall -O3 -ansi -x c++
4 #CXXGFLAGS = -Wall -O3
5 #CXXGFLAGS = -pg
6
7 .PHONY: std cphstl
8
9 all: std dynamic-array safe-dynamic-array hat safe-hat lap safe-lap
10
11 std:
12     $(CXX) $(CXXGFLAGS) -DSTD drive.c++
13     ./a.out
14
15 dynamic-array:
16     $(CXX) $(CXXGFLAGS) -DCPHSTL drive.c++
17     ./a.out
18
19 safe-dynamic-array:
20     $(CXX) $(CXXGFLAGS) -DSAFE_CPHSTL drive.c++
21     ./a.out
22
23 hat:
24     $(CXX) $(CXXGFLAGS) -DHAT_CPHSTL drive.c++
25     ./a.out
26
27 safe-hat:

```

```
28     $(CXX) $(CXXGFLAGS) -DSAFE_HAT_CPHSTL drive.c++
29     ./a.out
30
31 lap:
32     $(CXX) $(CXXGFLAGS) -DLAP_CPHSTL drive.c++
33     ./a.out
34
35 safe-lap:
36     $(CXX) $(CXXGFLAGS) -DSAFE_LAP_CPHSTL drive.c++
37     ./a.out
38
39 benz: push_back pop_back sequential-access random-access insert
40
41 push_back:
42     export PYTHONPATH=$(HOME)/CPHSTL/Tool/Benz; python plot-push-back.py
43
44 pop_back:
45     export PYTHONPATH=$(HOME)/CPHSTL/Tool/Benz; python plot-pop-back.py
46
47 sequential-access:
48     export PYTHONPATH=$(HOME)/CPHSTL/Tool/Benz; python plot-sequential-access.py
49
50 random-access:
51     export PYTHONPATH=$(HOME)/CPHSTL/Tool/Benz; python plot-random-access.py
52
53 insert:
54     export PYTHONPATH=$(HOME)/CPHSTL/Tool/Benz; python plot-insert.py
55
56 clean:
57     @-rm -rf *.pyc *~ a.out *~ std plot.[0-9]* driver.[0-9]* debug.[0-9]* /tmp/
        clock_overhead
```