"...one of the most highly regarded and expertly designed C++ library projects in the world."
— Herb Sutter and Andrei Alexandrescu, C++ Coding Standards

Performance

Back insertion and destruction
Reversing
Sorting
Write access
Conclusions

Boost.Intrusive containers offer speed improvements compared to non-intrusive containers primarily because:

- They minimize memory allocation/deallocation calls.
- They obtain better memory locality.

This section will show performance tests comparing some operations on `boost::intrusive::list` and `std::list`:

- Insertions using `push_back` and container destruction will show the overhead associated with memory allocation/deallocation.
- The `reverse` member function will show the advantages of the compact memory representation that can be achieved with intrusive containers.
- The `sort` and `write access` tests will show the advantage of intrusive containers minimizing memory accesses compared to containers of pointers.

Given an object of type `T`, `boost::intrusive::list<T>` can replace `std::list<T>` to avoid memory allocation overhead, or it can replace `std::list<T*>` when the user wants containers with polymorphic values or wants to share values between several containers. Because of this versatility, the performance tests will be executed for 6 different list types:

- 3 intrusive lists holding a class named `itest_class`, each one with a different linking policy (`normal_link`, `safe_link`, `auto_unlink`). The `itest_class` objects will be tightly packed in a `std::vector<itest_class>` object.
- `std::list<test_class>`, where `test_class` is exactly the same as `itest_class`, but without the intrusive hook.
- `std::list<test_class*>`. The list will contain pointers to `test_class` objects tightly packed in a `std::vector<test_class>` object. We'll call this configuration *compact pointer list*
- `std::list<test_class*>`. The list will contain pointers to `test_class` objects owned by a `std::list<test_class>` object. We'll call this configuration *disperse pointer list*.

Both `test_class` and `itest_class` are templatized classes that can be configured with a boolean to increase the size of the object. This way, the tests can be executed with small and big objects. Here is the first part of the testing code, which shows the definitions of `test_class` and `itest_class` classes, and some other utilities:

```
//Iteration and element count defines
const int NumIter = 100;
const int NumElements   = 50000;

using namespace boost::intrusive;

template<bool BigSize>  struct filler        {  int dummy[10];   };
template <>             struct filler<false> {};

template<bool BigSize> //The object for non-intrusive containers
struct test_class :  private filler<BigSize>
{
   int i_;
   test_class()               {}
   test_class(int i) :  i_(i) {}
   friend bool operator <(const test_class &l, const test_class &r)  {  return l.i_ < r.i_;  }
```

```
   friend bool operator >(const test_class &l, const test_class &r)  {  return l.i_ > r.i_;  }
};

template <bool BigSize, link_mode_type LinkMode>
struct itest_class   //The object for intrusive containers
   :  public list_base_hook<link_mode<LinkMode> >,  public test_class<BigSize>
{
   itest_class()                                  {}
   itest_class(int i) : test_class<BigSize>(i)  {}
};

template<class FuncObj> //Adapts functors taking values to functors taking pointers
struct func_ptr_adaptor  :  public FuncObj
{
   typedef typename FuncObj::first_argument_type*  first_argument_type;
   typedef typename FuncObj::second_argument_type* second_argument_type;
   typedef typename FuncObj::result_type           result_type;
   result_type operator()(first_argument_type a,  second_argument_type b) const
      {  return FuncObj::operator()(*a, *b); }
};
```

As we can see, `test_class` is a very simple class holding an `int`. `itest_class` is just a class that has a base hook (`list_base_hook`) and also derives from `test_class`.

`func_ptr_adaptor` is just a functor adaptor to convert function objects taking `test_list` objects to function objects taking pointers to them.

You can find the full test code code in the perf_list.cpp source file.

Back insertion and destruction

The first test will measure the benefits we can obtain with intrusive containers avoiding memory allocations and deallocations. All the objects to be inserted in intrusive containers are allocated in a single allocation call, whereas `std::list` will need to allocate memory for each object and deallocate it for every erasure (or container destruction).

Let's compare the code to be executed for each container type for different insertion tests:

```
std::vector<typename ilist::value_type> objects(NumElements);
ilist l;
for(int i = 0; i < NumElements; ++i)
   l.push_back(objects[i]);
//Elements are unlinked in ilist's destructor
//Elements are destroyed in vector's destructor
```

For intrusive containers, all the values are created in a vector and after that inserted in the intrusive list.

```
stdlist l;
for(int i = 0; i < NumElements; ++i)
   l.push_back(typename stdlist::value_type(i));
//Elements unlinked and destroyed in stdlist's destructor
```

For a standard list, elements are pushed back using push_back().

```
std::vector<typename stdlist::value_type> objects(NumElements);
stdptrlist l;
for(int i = 0; i < NumElements; ++i)
   l.push_back(&objects[i]);
//Pointers to elements unlinked and destroyed in stdptrlist's destructor
//Elements destroyed in vector's destructor
```

For a standard compact pointer list, elements are created in a vector and pushed back in the pointer list using push_back().

```
stdlist objects;  stdptrlist l;
for(int i = 0; i < NumElements; ++i){
   objects.push_back(typename stdlist::value_type(i));
   l.push_back(&objects.back());
}
//Pointers to elements unlinked and destroyed in stdptrlist's destructor
//Elements unlinked and destroyed in stdlist's destructor
```

For a *disperse pointer list*, elements are created in a list and pushed back in the pointer list using push_back().

These are the times in microseconds for each case, and the normalized time:

Table 10.2. Back insertion + destruction times for Visual C++ 7.1 / Windows XP

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	5000 / 22500	1 / 1
`safe_link` intrusive list	7812 / 32187	1.56 / 1.43
`auto_unlink` intrusive list	10156 / 41562	2.03 / 1.84
Standard list	76875 / 97500	5.37 / 4.33
Standard compact pointer list	76406 / 86718	15.28 / 3.85
Standard disperse pointer list	146562 / 175625	29.31 / 7.80

Table 10.3. Back insertion + destruction times for GCC 4.1.1 / MinGW over Windows XP

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	4375 / 22187	1 / 1
`safe_link` intrusive list	7812 / 32812	1.78 / 1.47
`auto_unlink` intrusive list	10468 / 42031	2.39 / 1.89
Standard list	81250 / 98125	18.57 / 4.42
Standard compact pointer list	83750 / 94218	19.14 / 4.24
Standard disperse pointer list	155625 / 175625	35.57 / 7.91

Table 10.4. Back insertion + destruction times for GCC 4.1.2 / Linux Kernel 2.6.18 (OpenSuse 10.2)

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	4792 / 20495	1 / 1
`safe_link` intrusive list	7709 / 30803	1.60 / 1.5
`auto_unlink` intrusive list	10180 / 41183	2.12 / 2.0
Standard list	17031 / 32586	3.55 / 1.58
Standard compact pointer list	27221 / 34823	5.68 / 1.69
Standard disperse pointer list	102272 / 60056	21.34 / 2.93

The results are logical: intrusive lists just need one allocation. The destruction time of the `normal_link` intrusive

container is trivial (complexity: `O(1)`), whereas `safe_link` and `auto_unlink` intrusive containers need to put the hooks of erased values in the default state (complexity: `O(NumElements)`). That's why `normal_link` intrusive list shines in this test.

Non-intrusive containers need to make many more allocations and that's why they lag behind. The `disperse pointer list` needs to make `NumElements*2` allocations, so the result is not surprising.

The Linux test shows that standard containers perform very well against intrusive containers with big objects. Nearly the same GCC version in MinGW performs worse, so maybe a good memory allocator is the reason for these excellent results.

Reversing

The next test measures the time needed to complete calls to the member function `reverse()`. Values (`test_class` and `itest_class`) and lists are created as explained in the previous section.

Note that for pointer lists, `reverse` **does not need to access `test_class` values stored in another list or vector**, since this function just needs to adjust internal pointers, so in theory all tested lists need to perform the same operations.

These are the results:

Table 10.5. Reverse times for Visual C++ 7.1 / Windows XP

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	2656 / 10625	1 / 1.83
`safe_link` intrusive list	2812 / 10937	1.05 / 1.89
`auto_unlink` intrusive list	2710 / 10781	1.02 / 1.86
Standard list	5781 / 14531	2.17 / 2.51
Standard compact pointer list	5781 / 5781	2.17 / 1
Standard disperse pointer list	10781 / 15781	4.05 / 2.72

Table 10.6. Reverse times for GCC 4.1.1 / MinGW over Windows XP

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	2656 / 10781	1 / 2.22
`safe_link` intrusive list	2656 / 10781	1 / 2.22
`auto_unlink` intrusive list	2812 / 10781	1.02 / 2.22
Standard list	4843 / 12500	1.82 / 2.58
Standard compact pointer list	4843 / 4843	1.82 / 1
Standard disperse pointer list	9218 / 12968	3.47 / 2.67

Table 10.7. Reverse times for GCC 4.1.2 / Linux Kernel 2.6.18 (OpenSuse 10.2)

Container	Time in us/iteration (small object /	Normalized time (small object /

	big object)	**big object)**
`normal_link` intrusive list	2742 / 10847	1 / 3.41
`safe_link` intrusive list	2742 / 10847	1 / 3.41
`auto_unlink` intrusive list	2742 / 11027	1 / 3.47
Standard list	3184 / 10942	1.16 / 3.44
Standard compact pointer list	3207 / 3176	1.16 / 1
Standard disperse pointer list	5814 / 13381	2.12 / 4.21

For small objects the results show that the compact storage of values in intrusive containers improve locality and reversing is faster than with standard containers, whose values might be dispersed in memory because each value is independently allocated. Note that the dispersed pointer list (a list of pointers to values allocated in another list) suffers more because nodes of the pointer list might be more dispersed, since allocations from both lists are interleaved in the code:

```
//Object list (holding `test_class`)
stdlist objects;

//Pointer list (holding `test_class` pointers)
stdptrlist l;

for(int i = 0; i < NumElements; ++i){
   //Allocation from the object list
   objects.push_back(stdlist::value_type(i));
   //Allocation from the pointer list
   l.push_back(&objects.back());
}
```

For big objects the compact pointer list wins because the reversal test doesn't need access to values stored in another container. Since all the allocations for nodes of this pointer list are likely to be close (since there is no other allocation in the process until the pointer list is created) locality is better than with intrusive containers. The dispersed pointer list, as with small values, has poor locality.

Sorting

The next test measures the time needed to complete calls to the member function `sort(Pred pred)`. Values (`test_class` and `itest_class`) and lists are created as explained in the first section. The values will be sorted in ascending and descending order each iteration. For example, if *l* is a list:

```
for(int i = 0; i < NumIter; ++i){
   if(!(i % 2))
      l.sort(std::greater<stdlist::value_type>());
   else
      l.sort(std::less<stdlist::value_type>());
}
```

For a pointer list, the function object will be adapted using `func_ptr_adaptor`:

```
for(int i = 0; i < NumIter; ++i){
   if(!(i % 2))
      l.sort(func_ptr_adaptor<std::greater<stdlist::value_type> >());
   else
      l.sort(func_ptr_adaptor<std::less<stdlist::value_type> >());
}
```

Note that for pointer lists, `sort` will take a function object that **will access `test_class` values stored in another list or vector**, so pointer lists will suffer an extra indirection: they will need to access the `test_class` values stored in another container to compare two elements.

These are the results:

Table 10.8. Sort times for Visual C++ 7.1 / Windows XP

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	16093 / 38906	1 / 1
`safe_link` intrusive list	16093 / 39062	1 / 1
`auto_unlink` intrusive list	16093 / 38906	1 / 1
Standard list	32343 / 56406	2.0 / 1.44
Standard compact pointer list	33593 / 46093	2.08 / 1.18
Standard disperse pointer list	46875 / 68593	2.91 / 1.76

Table 10.9. Sort times for GCC 4.1.1 / MinGW over Windows XP

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	15000 / 39218	1 / 1
`safe_link` intrusive list	15156 / 39531	1.01 / 1.01
`auto_unlink` intrusive list	15156 / 39531	1.01 / 1.01
Standard list	34218 / 56875	2.28 / 1.45
Standard compact pointer list	35468 / 49218	2.36 / 1.25
Standard disperse pointer list	47656 / 70156	3.17 / 1.78

Table 10.10. Sort times for GCC 4.1.2 / Linux Kernel 2.6.18 (OpenSuse 10.2)

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	18003 / 40795	1 / 1
`safe_link` intrusive list	18003 / 41017	1 / 1
`auto_unlink` intrusive list	18230 / 40941	1.01 / 1
Standard list	26273 / 49643	1.45 / 1.21
Standard compact pointer list	28540 / 43172	1.58 / 1.05
Standard disperse pointer list	35077 / 57638	1.94 / 1.41

The results show that intrusive containers are faster than standard containers. We can see that the pointer list holding pointers to values stored in a vector is quite fast, so the extra indirection that is needed to access the value is minimized because all the values are tightly stored, improving caching. The disperse list, on the other hand, is slower because the indirection to access values stored in the object list is more expensive than accessing values stored in a vector.

Write access

The next test measures the time needed to iterate through all the elements of a list, and increment the value of the internal `i_` member:

```
stdlist::iterator it(l.begin()), end(l.end());
for(; it != end; ++it)
   ++(it->i_);
```

Values (`test_class` and `itest_class`) and lists are created as explained in the first section. Note that for pointer lists, the iteration will suffer an extra indirection: they will need to access the `test_class` values stored in another container:

```
stdptrlist::iterator it(l.begin()), end(l.end());
for(; it != end; ++it)
   ++((*it)->i_);
```

These are the results:

Table 10.11. Write access times for Visual C++ 7.1 / Windows XP

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	2031 / 8125	1 / 1
`safe_link` intrusive list	2031 / 8281	1 / 1.01
`auto_unlink` intrusive list	2031 / 8281	1 / 1.01
Standard list	4218 / 10000	2.07 / 1.23
Standard compact pointer list	4062 / 8437	2.0 / 1.03
Standard disperse pointer list	8593 / 13125	4.23 / 1.61

Table 10.12. Write access times for GCC 4.1.1 / MinGW over Windows XP

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	2343 / 8281	1 / 1
`safe_link` intrusive list	2500 / 8281	1.06 / 1
`auto_unlink` intrusive list	2500 / 8281	1.06 / 1
Standard list	4218 / 10781	1.8 / 1.3
Standard compact pointer list	3906 / 8281	1.66 / 1
Standard disperse pointer list	8281 / 13750	3.53 / 1.66

Table 10.13. Write access times for GCC 4.1.2 / Linux Kernel 2.6.18 (OpenSuse 10.2)

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	2286 / 8468	1 / 1.1
`safe_link` intrusive list	2381 / 8412	1.04 / 1.09
`auto_unlink` intrusive list	2301 / 8437	1.01 / 1.1
Standard list	3044 / 9061	1.33 / 1.18
Standard compact pointer list	2755 / 7660	1.20 / 1
Standard disperse pointer list	6118 / 12453	2.67 / 1.62

As with the read access test, the results show that intrusive containers outperform all other containers if the values are tightly packed in a vector. The disperse list is again the slowest.

Conclusions

Intrusive containers can offer performance benefits that cannot be achieved with equivalent non-intrusive containers. Memory locality improvements are noticeable when the objects to be inserted are small. Minimizing memory allocation/deallocation calls is also an important factor and intrusive containers make this simple if the user allocates all the objects to be inserted in intrusive containers in containers like `std::vector` or `std::deque`.