This talk is about the heaps we all love. I will explain how the heap functions are implemented in the CPH STL program library. The main contribution of the work done by my co-workers and myself is an experimental evaluation of various heap variants proposed in the computing literature. We have also done micro-benchmarking which gives some directions for future research.

These slides are available at http://www.cphstl.dk/.
9th Scandinavian Workshop on Algorithm Theory

July 8–10, 2004
Louisiana Museum of Modern Art
Humlebæk, Denmark
http://swat.diku.dk/

Deadline for submission:
February 10, 2004 at noon (GMT)

Notification of authors:
March 23, 2004

Final version due:
April 20, 2004

End of early registration:
May 4, 2004
Mission

The Standard Template Library, or STL for short, is a library of algorithms and data structures that has been incorporated into the C++ language standard and now ships with all modern C++ compilers.

In existing STL implementations many STL components are tuned for performance, some even outdoing most of their hand-crafted counterparts. In many cases, however, there is still room for improvements. The CPH STL aims to improve both algorithmic and an implementational level as suggested, from our ongoing research conducted here at DIKU (Performance Engineering Group).

The purpose of this project is:

- to study and analyse existing specifications for and implementations of STL to determine the best approaches to optimization
- to design alternative/enhanced versions of individual STL components using standard algorithmic and performance engineering techniques
- to implement and document the new versions in C++

Those will be the main strands of activity, but the project is a good exercise in software development using up-to-date methods.

--

Last modified April 8, 2003 10:15:36 AM
Heap functions in the STL

```c
void push_heap(position A, position Z, ordering f);
```

**Effect:**

![Effect diagram](image1)

at most \(\log_2 n\) comparisons

```c
void pop_heap(position A, position Z, ordering f);
```

**Effect:**

![Effect diagram](image2)

at most \(2\log_2 n\) comparisons

```c
void make_heap(position A, position Z, ordering f);
```

**Effect:**

![Effect diagram](image3)

at most \(3n\) comparisons

```c
void sort_heap(position A, position Z, ordering f);
```

**Effect:**

![Effect diagram](image4)

at most \(n \log_2 n\) comparisons

© Performance Engineering Laboratory
How would you do it?
Operation sequence (hold model):

\[
push()^N [pop() push()]^K
\]

\[e \leftarrow pop()\]

increase the priority of \(e\) by \(-\ln(\text{drand}())\)

\[push(e)\]

Input data:

- element size: 4 B; \#elements: 1–2^{13.5}

Environment:

- computer: VAX 11/780 running UNIX (BSD 4.2);
- cache: 8 kB; TLB: 64 entries; compiler: Berkeley Pascal with optimization enabled
Operation sequence:
  Hold model?
  #define NOTSORANDNUM(x) (x + RANDNUM())

Input data:
  element size: 8 B; #elements: $2^{10} - 2^{23}$

Environment:
  computer: DEC Alphastation 250; processor: Alpha 21064A 266 MHz; L1 cache: 8 kB; L2 cache: direct-mapped, 2 MB, 32 B per line; compiler?: cc
Operation sequence:

\[ [\text{push()} \text{pop()} \text{push()}]^N [\text{pop()} \text{push()} \text{pop()}]^N \]

Input data:
- element size: 4 B, drawn randomly; satellite data: 4 B; #elements: \(2^8 - 2^{23}\)

Environment:
- computer: Pentium II 300 MHz; compiler g++ -O6
Brengel et al. 1999

Operation sequence:
\[ \text{push}(N)^N / \text{pop}(N)^N \]

Input data:
- element size: 4 B, drawn randomly from \([0..10^7]\);
- \#elements: \(1 \cdot 10^6 – 200 \cdot 10^6\)

Environment:
- computer: Sparc Ultra 1/143; main memory: 256 MB, 8 kB per page; local disk: 9 GB fastwide SCSI; logical block size: 64 kB; buffer size: 16 MB
**Operation sequence:**

\[ \text{make}(N)[\text{pop}()]^N \]

**Input data:**

- element size: 4 B, floating point numbers drawn randomly;
- \#elements: \(10^6\); ordering: \(f^0(x) = x\) and \(f^i(x) = \ln(f^{i-1}(x+1))\) for \(i > 0\)

**Environment:**

- computer: Pentium III 450 MHz; compiler `g++ -O2`

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>(f^0)</th>
<th>(f^1)</th>
<th>(f^2)</th>
<th>(f^3)</th>
<th>(f^4)</th>
<th>(f^5)</th>
<th>(f^6)</th>
<th>(f^7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUICKSORT</td>
<td>3.86</td>
<td>14.59</td>
<td>26.73</td>
<td>39.04</td>
<td>51.47</td>
<td>63.44</td>
<td>75.68</td>
<td>87.89</td>
</tr>
<tr>
<td>CLEVER-QUICKSORT</td>
<td>3.56</td>
<td>12.84</td>
<td>23.67</td>
<td>33.16</td>
<td>43.80</td>
<td>54.37</td>
<td>64.62</td>
<td>75.08</td>
</tr>
<tr>
<td>BOTTOM-UP-HEAPSORT</td>
<td>5.73</td>
<td>13.49</td>
<td>22.60</td>
<td>32.05</td>
<td>41.59</td>
<td>51.14</td>
<td>60.62</td>
<td>70.11</td>
</tr>
<tr>
<td>MDR-HEAPSORT</td>
<td>7.14</td>
<td>15.39</td>
<td>24.37</td>
<td>33.82</td>
<td>43.04</td>
<td>52.63</td>
<td>61.87</td>
<td>71.02</td>
</tr>
<tr>
<td>WEAK-HEAPSORT</td>
<td>7.15</td>
<td>14.89</td>
<td>23.66</td>
<td>32.82</td>
<td>41.97</td>
<td>51.08</td>
<td>60.13</td>
<td>69.27</td>
</tr>
<tr>
<td>RELAXED-WEAK-HEAPSORT</td>
<td>8.29</td>
<td>15.96</td>
<td>24.63</td>
<td>33.51</td>
<td>42.32</td>
<td>51.33</td>
<td>60.06</td>
<td>68.83</td>
</tr>
<tr>
<td>GREEDY-WEAK-HEAPSORT</td>
<td>9.09</td>
<td>16.60</td>
<td>25.24</td>
<td>34.12</td>
<td>43.03</td>
<td>51.77</td>
<td>60.72</td>
<td>69.78</td>
</tr>
<tr>
<td>QUICK-HEAPSORT</td>
<td>6.35</td>
<td>15.89</td>
<td>26.20</td>
<td>36.98</td>
<td>47.59</td>
<td>58.16</td>
<td>69.37</td>
<td>79.59</td>
</tr>
<tr>
<td>QUICK-WEAK-HEAPSORT</td>
<td>6.08</td>
<td>14.49</td>
<td>23.88</td>
<td>33.49</td>
<td>43.30</td>
<td>52.99</td>
<td>62.85</td>
<td>72.54</td>
</tr>
<tr>
<td>CLEVER-HEAPSORT</td>
<td>5.30</td>
<td>14.01</td>
<td>23.65</td>
<td>33.65</td>
<td>43.95</td>
<td>53.79</td>
<td>63.60</td>
<td>73.94</td>
</tr>
<tr>
<td>CLEVER-WEAK-HEAPSORT</td>
<td>5.97</td>
<td>13.82</td>
<td>22.83</td>
<td>31.95</td>
<td>41.31</td>
<td>50.40</td>
<td>59.58</td>
<td>69.61</td>
</tr>
</tbody>
</table>
How would you do it now?
Sanders’ programs on Pentium II

Execution time per element [in nanoseconds]

Sanders’ programs: [push()] N [pop()] N

2-ary heap
4-ary heap

Sanders’ programs on Pentium II: [push() pop() push()] N [pop() push() pop()] N
Sanders' programs on Pentium III:

- `push()` $N$
- `pop()` $N$

2-ary heap
4-ary heap

Execution time per element [in nanoseconds]
Sanders’ programs on Pentium IV

Execution time per element [in nanoseconds]
## Cost of unsigned int operations

<table>
<thead>
<tr>
<th>initializations</th>
<th>instruction</th>
<th>unsigned int</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p \leftarrow 1 )  \  ( a[i] \leftarrow 0 )  \  ( x \leftarrow 2^{20} )</td>
<td>( a[i] \leftarrow x )</td>
<td>( n = 2^{10} \ldots 2^{24} ) 4.1–4.7 ns</td>
</tr>
<tr>
<td>( p \leftarrow 617 )  \  ( a[i] \leftarrow 0 )  \  ( x \leftarrow 2^{20} )</td>
<td>( a[i] \leftarrow x )</td>
<td>( n = 2^{10} \ldots 2^{14} ) 7.3–8.9 ns ( n = 2^{15} ) 12 ns ( n = 2^{16} ) 29 ns ( n = 2^{16} \ldots 2^{22} ) 62–63 ns</td>
</tr>
<tr>
<td>( p \leftarrow 1 )  \  ( a[i] \leftarrow 0 )  \  ( x \leftarrow 2^{20} )</td>
<td>( x \leftarrow a[i] )</td>
<td>( n = 2^{10} \ldots 2^{24} ) 3.3–3.8 ns</td>
</tr>
<tr>
<td>( p \leftarrow 617 )  \  ( a[i] \leftarrow 0 )  \  ( x \leftarrow 2^{20} )</td>
<td>( x \leftarrow a[i] )</td>
<td>( n = 2^{10} \ldots 2^{15} ) 3.3–4.1 ns ( n = 2^{16} ) 23 ns ( n = 2^{17} \ldots 2^{22} ) 45–55 ns</td>
</tr>
<tr>
<td>( p \leftarrow 1 )  \  ( a[i] \leftarrow 0 )  \  ( x \leftarrow 2^{20} )</td>
<td>( r \leftarrow (a[i] &lt; x) )</td>
<td>( n = 2^{10} \ldots 2^{24} ) 5.3–5.8 ns</td>
</tr>
<tr>
<td>( p \leftarrow 1 )  \  ( a[i] \leftarrow 0 )  \  ( x \leftarrow 2^{20} )</td>
<td>( r \leftarrow (\ln(a[i]) &lt; \ln(x)) )</td>
<td>( n = 2^{10} \ldots 2^{24} ) 580–610 ns</td>
</tr>
</tbody>
</table>
### Cost of bigint operations

<table>
<thead>
<tr>
<th>initializations</th>
<th>instruction</th>
<th>bigint</th>
</tr>
</thead>
</table>
| \( p \leftarrow 1 \)  
\( a[i] \leftarrow 0 \)  
\( x \leftarrow 2^{20} \) | \( a[i] \leftarrow x \) | \( n = 2^{10} \ldots 2^{21} \)  
60–66 ns  
290 ns |
| \( p \leftarrow 617 \)  
\( a[i] \leftarrow 0 \)  
\( x \leftarrow 2^{20} \) | \( a[i] \leftarrow x \) | \( n = 2^{10} \ldots 2^{12} \)  
75–78 ns  
117 ns  
229 ns  
297–318 ns  
748–752 ns |
| \( p \leftarrow 1 \)  
\( a[i] \leftarrow 0 \)  
\( x \leftarrow 2^{20} \) | \( x \leftarrow a[i] \) | \( n = 2^{10} \ldots 2^{22} \)  
18–21 ns |
| \( p \leftarrow 617 \)  
\( a[i] \leftarrow 0 \)  
\( x \leftarrow 2^{20} \) | \( x \leftarrow a[i] \) | \( n = 2^{10} \ldots 2^{12} \)  
24 ns  
83 ns  
180 ns  
230–260 ns |
| \( p \leftarrow 1 \)  
\( a[i] \leftarrow 0 \)  
\( x \leftarrow 2^{20} \) | \( r \leftarrow (a[i] < x) \) | \( n = 2^{10} \ldots 2^{22} \)  
13–16 ns |
Other current research

**Pointer-based methods:**

- hopelessly slow
  \[\rightarrow\] theoretical computer science

**Methods with good amortized bounds:**

- terrible worst case
  \[\rightarrow\] not relevant for us

**Methods with few element moves:**

- bad cache behaviour
  \[\rightarrow\] not good for us

**External-memory methods:**

- high constants
  \[\rightarrow\] relevant only for very large data sets

**Cache-oblivious methods:**

- huge constants
  \[\rightarrow\] theoretical computer science
Our policy-based framework

template <arity d, typename position, typename ordering>
class heap_policy {
public:
    typedef typename
        std::iterator_traits<position>::difference_type index;
    typedef typename
        std::iterator_traits<position>::difference_type level;
    typedef typename
        std::iterator_traits<position>::value_type element;

    template <typename integer>
    heap_policy(integer n = 0);

    bool is_root(index) const;
    bool is_first_child(index) const;
    index size() const;
    level depth(index) const;
    index root() const;
    index leftmost_leaf() const;
    index last_leaf() const;
    index first_child(index) const;
    index parent(index) const;
    index ancestor(index, level) const;
    index top_some_absent(position, index,
        const ordering&) const;
    index top_all_present(position, index,
        const ordering&) const;
    void update(position, index, const element&);
    void erase_last_leaf(position, const ordering&);
    void insert_new_leaf(position, const ordering&);
private:
    index n;
};
## Input data

<table>
<thead>
<tr>
<th></th>
<th>cheap move</th>
<th>expensive move</th>
</tr>
</thead>
<tbody>
<tr>
<td>cheap comparison</td>
<td>unsigned int</td>
<td>bigint</td>
</tr>
<tr>
<td>expensive comparison</td>
<td>unsigned int</td>
<td>(int, bigint)</td>
</tr>
</tbody>
</table>
One new old idea: local heaps
Our solution for sort_heap()

In-place mergesort by Katajainen, Pasanen, and Teuhola [1996]

Fine-tuning not yet implemented

Almost as fast as quicksort, see CPH STL Report 2003-2
Our solution for make_heap()

Depth-first heap construction by Bojesen, Katajainen, and Spork [2000]

Almost optimal in all respects

Other work:

less element comparisons

→ theoretical computer science
Various approaches for pop_heap()

- top-down → many element comparisons
- bottom-up → typical case good
- move-saving bottom-up → theoretical computer science
- binary-search top-down
- two-levels-at-a-time top-down
Various approaches for \texttt{push\_heap()}

- move-saving top-down $\rightarrow$ slow
- bottom-up $\rightarrow$ typical case good
- bottom-up with buffering $\rightarrow$ complicated
- binary-search bottom-up
Efficiency of 2-, 3-, 4-ary heaps

Execution time per element [in nanoseconds]

Efficiency of various sorting functions for random integers

SGI::partial_sort()
Bottom-up approach: 3-ary heap
Bottom-up approach: 2-ary heap
Bottom-up approach: 4-ary heap
SGI::sort()
Efficiency of 2-, 3-, 4-ary heaps

Execution time per element [in nanoseconds]

Efficiency of various sorting functions for random integers using ln comparison

Bottom-up approach: 4-ary heap
SGI::sort()
Bottom-up approach: 3-ary heap
Bottom-up approach: 2-ary heap
SGI::partial_sort()
Efficiency of local heaps

Execution time per element [in nanoseconds]

Efficiency of various sorting functions for random integers
- SGI::partial_sort()
- Two-by-two top-down approach: 5-local heap
- Two-by-two top-down approach: 4-local heap
- Two-by-two top-down approach: 3-local heap
- Two-by-two top-down approach: 2-local heap
- Two-by-two top-down approach: 1-local heap
- SGI::sort()

Efficiency of various sorting functions for random bigints
- SGI::partial_sort()
- Two-by-two top-down approach: 5-local heap
- Two-by-two top-down approach: 4-local heap
- Two-by-two top-down approach: 3-local heap
- Two-by-two top-down approach: 2-local heap
- Two-by-two top-down approach: 1-local heap
- SGI::sort()
Efficiency of local heaps

Efficiency of various sorting functions for random integers using ln comparison

Two-by-two top-down approach: 1-local heap
Two-by-two top-down approach: 2-local heap
Two-by-two top-down approach: 3-local heap
Two-by-two top-down approach: 4-local heap
Two-by-two top-down approach: 5-local heap
SGI::sort() function: partial_sort source: SGI

Efficiency of various sorting functions for pairs of integers and bigints using ln comparison

Two-by-two top-down approach: 1-local heap
Two-by-two top-down approach: 2-local heap
Two-by-two top-down approach: 3-local heap
Two-by-two top-down approach: 4-local heap
Two-by-two top-down approach: 5-local heap
SGI::partial_sort()
Conclusions

- In 40 years — not much progress
- At the moment it is not clear how big the overhead of local heaps is for small problem sizes.
- Some combinations of various approaches have still to be tested.
- Code-tuning of the best approaches is still to be done.
- It takes time to develop fast library routines.
- How does technology influence on the efficiency of the library routines?
Exercise of the week

How many element comparisons incur the operation sequence

\[ [push() \mid pop()]^N \]

in the worst case? Or what is the amortized complexity of each of these operations?

1.5\(N \log_2 N\) is an obvious upper bound and \(N \log_2 N\) an obvious lower bound.

Recall that the operation sequence

\[ make(N)[pop()]^N \]

requires about 1.5\(N \log_2 N\) element comparisons.