# Node allocation in the STL

## Improving the memory usage

Marcelo Zimbres

Code: https://github.com/mzimbres/rtcpp

# Goals - Target audience

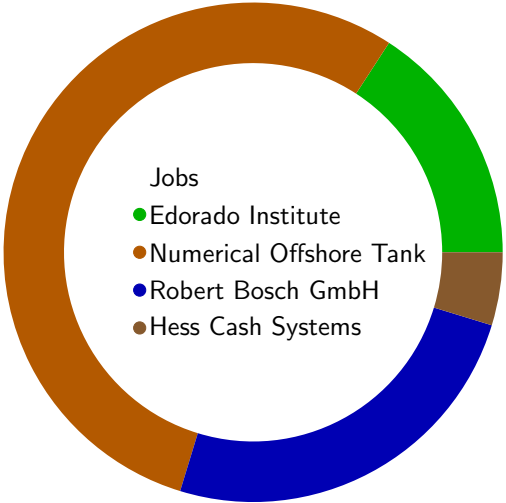
## Goals

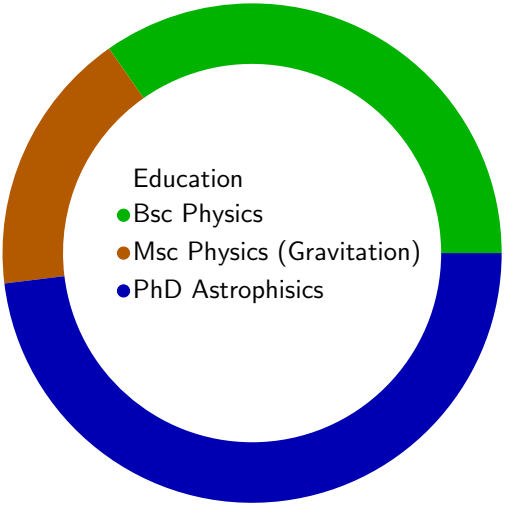
- Provide building blocks for memory allocation.
- Profit from static type information.
- Alternative to pointer chasing on traversal.
- Reduce pointer overhead in nodes.
- Allow fine tunning and reduce complexity.
- Reduce centralization.

## Target audience

- Embedded developers.
- Resource constrained systems.
- High performance.
- 24/7 availability.

# Who I am


Jobs
Edorado Institute
Numerical Offshore Tank
Robert Bosch GmbH
Hess Cash Systems


Education
Bsc Physics
Msc Physics (Gravitation)
PhD Astrophisics

# Outline


1. STL and memory management
   - Dynamic memory allocations in C++
   - Allocation patterns
   - Typical allocation goals and strategies
2. Supporting node-allocation in the STL
   - Proposal
   - Traversal without pointer chasing
   - Reducing the pointer overhead in nodes

# Memory allocation in C++

Note: I will use *Heap* to refer to free storage memory.

## Memory segments

- Data: Global and static variables.
- Stack: Restricted lifetime, small.
- Heap: No restriction on lifetime and size.

```
// Allocation in the data segment.
static int a; // Deallocation at program exit

// Allocation in the function stack frame.
int b; // Deallocation upon function return.

// Dynamic allocation on the heap.
auto* p = new int;

delete p; // Deallocation is not automatic.
```

## Inside STL containers

- Idea of segments abstracted away.
- Memory requested from allocators.
- Provide a customization point.

```
std::allocator<int> alloc; // Can have state.

// Requests enough space for one int.
auto* p = alloc.allocate(1);

// Returns memory to the allocator.
alloc.deallocate(p, 1);

// See also std::allocator_traits.
```

# Terminology used along this presentation

## Node vs. array allocation

### Node allocation

I will use **node allocation** to refer to allocations that have always the same size.

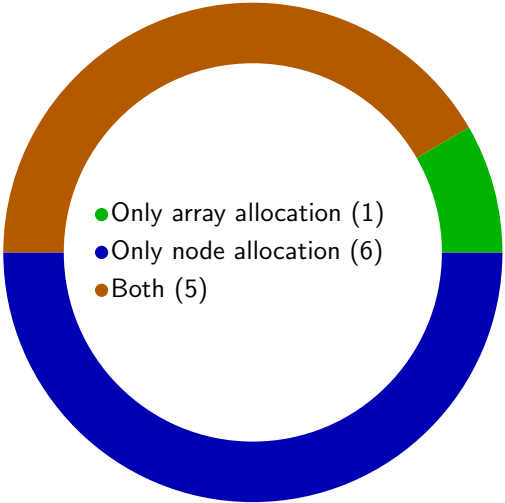
### Array allocation

For allocations with varying sizes I will use the term **array allocation**.

```
// Allocations inside std::list are one
// element at time (constant size).
auto* p1 = alloc.allocate(1);
auto* p2 = alloc.allocate(1);
auto* p3 = alloc.allocate(1);
auto* p4 = alloc.allocate(1);
```

```
// Allocations inside std::vector have
// increasing sizes (e.g. push_back).
auto* p1 = alloc.allocate(2);
auto* p2 = alloc.allocate(4);
auto* p3 = alloc.allocate(8);
auto* p4 = alloc.allocate(16);
```

# Containers and their allocation patterns




## STL containers (12)

- `std::vector`
- `std::list`
- `std::forward_list`
- `std::set`
- `std::multiset`
- `std::map`
- `std::multimap`
- `std::deque`
- `std::unordered_set`
- `std::unordered_multiset`
- `std::unordered_map`
- `std::unordered_multimap`

# Node allocation illustration

## How it is implemented in practice: pooling

### Procedure

1. Get a continuous block of memory with enough space for $n$ nodes.
2. Divide in blocks, link them as a stack and repeat if ever run out of memory.
3. After some allocations and deallocations.

Block with enough space for $n$ nodes

# Node allocation illustration

How it is implemented in practice: pooling

## Procedure

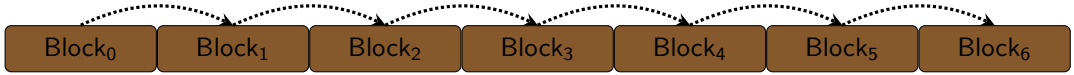
1. Get a continuous block of memory with enough space for $n$ nodes.
2. Divide in blocks, link them as a stack and repeat if ever run out of memory.
3. After some allocations and deallocations.

# Node allocation illustration

## How it is implemented in practice: pooling

### Procedure

1. Get a continuous block of memory with enough space for $n$ nodes.
2. Divide in blocks, link them as a stack and repeat if ever run out of memory.
3. After some allocations and deallocations.

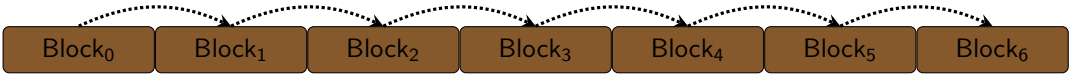


# Node allocation illustration

How it is implemented in practice: pooling

## Procedure

1. Get a continuous block of memory with enough space for $n$ nodes.
2. Divide in blocks, link them as a stack and repeat if ever run out of memory.
3. After some allocations and deallocations.



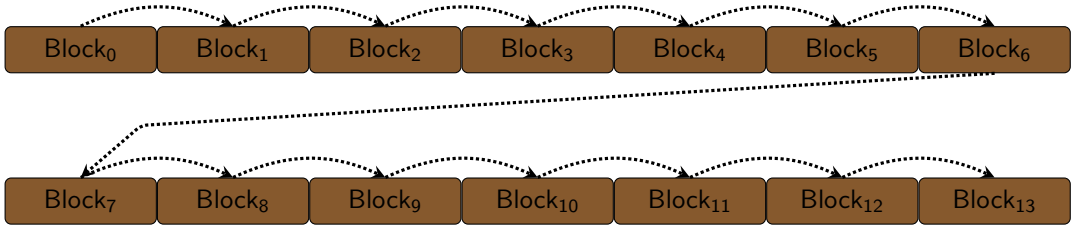

Run out of memory? Get another block.

# Node allocation illustration

How it is implemented in practice: pooling

## Procedure

1. Get a continuous block of memory with enough space for $n$ nodes.
2. Divide in blocks, link them as a stack and repeat if ever run out of memory.
3. After some allocations and deallocations.

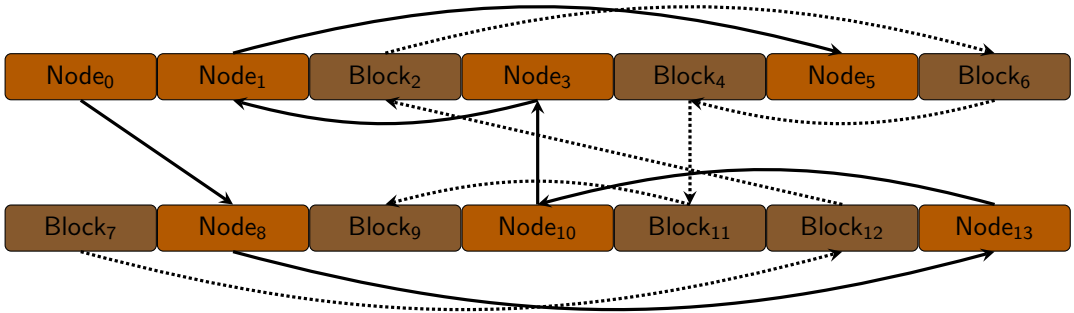


# Node allocation illustration

How it is implemented in practice: pooling

## Procedure

1. Get a continuous block of memory with enough space for $n$ nodes.
2. Divide in blocks, link them as a stack and repeat if ever run out of memory.
3. After some allocations and deallocations.



# Why the current allocator interface is suboptimal

## A closer look at allocators

```
pointer allocate(size_type n, ...);
```

```
void deallocate( pointer p
               , size_type n);
```

### Allocators and size management

- Allocation size is a runtime variable.
- Node allocation is however known at compile time.
- Blesses array allocation. Unaware of node allocation.
- No simple implementation. Has to handle any sizes.
- Leads to overly complex strategies.
- Leads to overheads like bookkeeping information.
- Encourages centralization.

# Simplicity matters

## How node allocation is implemented

```
pointer allocate_node()
{
 pointer q = avail; // The next free node
 if (avail)
   avail = avail->next;

 return q;
}

void deallocate_node(pointer p)
{
 p->next = avail;
 avail = p;
}
```

### Clear behaviour

Once you have nodes linked as a stack, allocation and deallocation reduces to these functions.

```
// Boost interface. Allows choosing the number
// of nodes per blocks. You know its best value
// better than malloc does.
node_allocator< int
              , 256 // Nodes per block
              > alloc;
```

# Outline


1. STL and memory management
   - Dynamic memory allocations in C++
   - Allocation patterns
   - Typical allocation goals and strategies
2. Supporting node-allocation in the STL
   - Proposal
   - Traversal without pointer chasing
   - Reducing the pointer overhead in nodes

# Typical allocation goals and strategies

`malloc`, `jemalloc`, `tcmalloc`, `nedmalloc`, `hoard`, etc.

## Typical goals and strategies

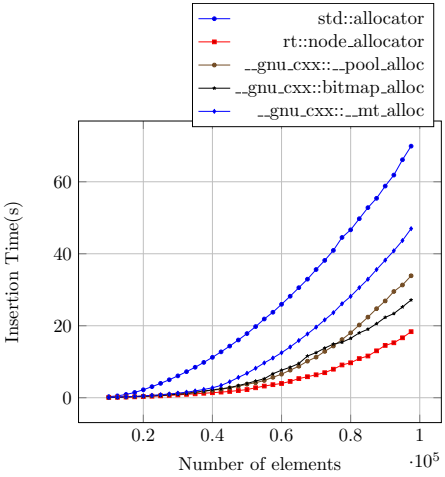
- Different strategy for small and big sizes.
- Small sizes: Many pre-allocated blocks.
- Large sizes: Best fit, rounding on memory pages.
- Avoid fragmentation. Improve locality.
- Avoid serialization on concurrent calls.
- Thread local buffers.
- Bookkeeping information on blocks.
- Rounding allocation sizes to fit pre-allocated buffers.
- Surprising (upsetting) behaviour.

## Centralization is bad

- Which strategy is better for my use case?
- Bookkeeping information, rounding, size management are unnecessary on node allocation
- Unclear behaviour for long running programs.
- Compile time information should not be thrown away.
- Node allocation is a building block.

# Surprises in allocator behaviour

Benchmarking `std::unordered_set` with well known allocators



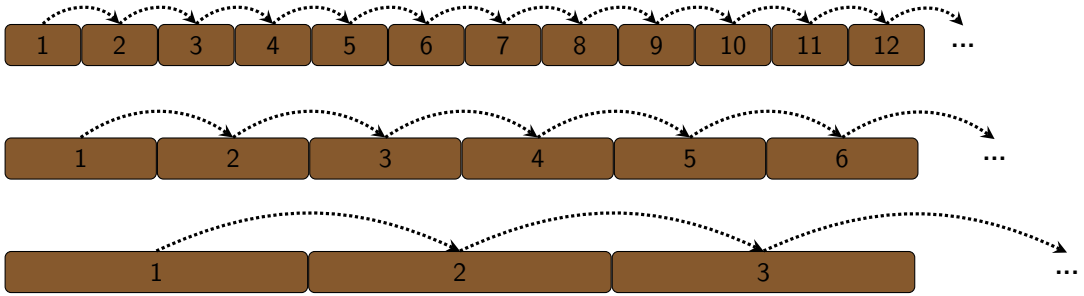

## Why `std::allocator` performs poorly?

- Difficult to say.
- Bookkeeping fields is making nodes bigger?
- Suboptimal rounding and size management?
- Should perform the same had node allocation been used internally.
- Scenario: Messy heap.
- This is unacceptable behaviour on many systems.

# Pools of blocks with different sizes

## Shouldn't need this if compile time information had been used.

- Many pre-allocated sizes. Possibly thread local.
- Small sizes are rounded to one of these.
- `std::malloc` has up to 170 of these lists.



## Node allocation is a building block

I've been arguing that node allocation is an important concept and deserves proper support in the STL. Now lets us see how we can get it in the STL and what further benefits it brings.

# Outline

1. STL and memory management
   - Dynamic memory allocations in C++
   - Allocation patterns
   - Typical allocation goals and strategies
2. Supporting node-allocation in the STL
   - Proposal
   - Traversal without pointer chasing
   - Reducing the pointer overhead in nodes

# Splitting *node* and *array* allocation in allocators

**Document number**: P0310R0
**Date**: 2016-03-19
**Project**: Programming Language C++
**Audience**: Library Evolution Working Group
**Reply to**: Marcelo Zimbres (mzimbres@gmail.com)

**Abstract:** This is a non-breaking proposal to the C++ standard that aims to reduce allocator complexity, support realtime allocation and improve performance of node-based containers by making a clear distinction between *node* and *array* allocation in the `std::allocator_traits` interface. Two new member functions are proposed, `allocate_node` and `deallocate_node`. We also propose that the container node type should be exposed to the user. A prototype implementation is provided.

# Node allocation support

Two new member functions in `std::allocator_traits`

```
// Boost interface (thanks to Ion Gaztanaga for
// suggesting it).
template<class Alloc>
struct allocator_traits {
  // Calls a.allocate_node() if present otherwise calls
  // Alloc::allocate(1). Memory allocated with this
  // function must be deallocated with deallocate_node.
  pointer allocate_node(Alloc& a);

  // Calls a.deallocate_node(pointer) if present
  // otherwise calls Alloc::deallocate(p, 1). Can only
  // be used with memory allocated with allocate_node.
  void deallocate_node(Alloc& a, pointer p);
};
```

## Why

- No size management.
- Well defined behaviour.
- Names reflect intention.
- No surprises

# Containers that perform both *node* and *array* allocation

## Exposing the node type

### Problem

Which allocator is the node allocator?

```
// Allocator instance does not know which
// type it will serve.
my_alloc<int> alloc;

// Rebinds to two further allocator types
// and construct them from alloc.
std::unordered_set< int, ...
                  , my_alloc<int>> obj(alloc);

// Triggers both node and array allocation.
// Which allocator will have n = 1 is unknown.
obj.insert(10);
```

### Fix

Let the allocator know the node type.

```
using node_type =
    std::unordered_set<int>::node_type

// Allocator knows the node type and offers
// only allocate_node() for node allocator.
// (i.e. disables allocate(n)).
my_alloc<int, node_type> alloc;

std::unordered_set< int
                  , my_alloc<int, node_type>
                  > obj(alloc);

// No need for testing n == 1.
obj.insert(10);
```

# The node type is not independent of the allocator

## Fancy pointers

### Typical node

No customization of the pointer type.

```
template <class T>
struct node {
  T info;
  node* next;
};
```

### STL node

Allows customization of the pointer type.

```
template < class T
         , class Ptr // Defined by the allocator
         >
struct node {
  T info;
  using pointer = typename
      std::pointer_traits<Ptr>::template
          rebind<node<T, Ptr>>;
  pointer next;
};
```

# We need to rebind the node type

Node interface

## Circular type definition

- The node is unknown until the container is defined.
- The container is unknown until the allocator is defined.
- The allocator unknown until the node is be defined.
- Solution: Offer a rebind function.

```
// Rebinding allows changing the node pointer
// type. Allocator uses it to define the node
// with its own pointer type.
template <class T, class Ptr>
struct node {
  ...
  template<class U, class K>
  struct rebind { using other = node<U , K>; };
};
```

```
// Gets the node type with any allocator.
using node_type = typename
    std::list<int>::node_type.

// Allocator rebinds the node type internally.
// to whatever pointer type it uses.
node_allocator<int, node_type, 256> alloc;
```

# Simple use case

## Containers with small number of elements

```
std::aligned_storage< sizeof(node_type)
                    , alignof(node_type)
                    >::type buffer[64];

// Allocator with very simple implementation.
node_allocator< int, node_type
              , 64> alloc(buffer);

std::list<int , node_allocator<...>
    > l(alloc);

// Allocation and deallocation implemented
// with 6 lines of code.
l = {27, 1, 60};
```

### Thoughts

- Node sizes are known at compile time.
- Stores them in the stack.
- Minimum amount of memory needed for $n$ elements.
- Never touches the heap.
- Let us move to more interesting usages.

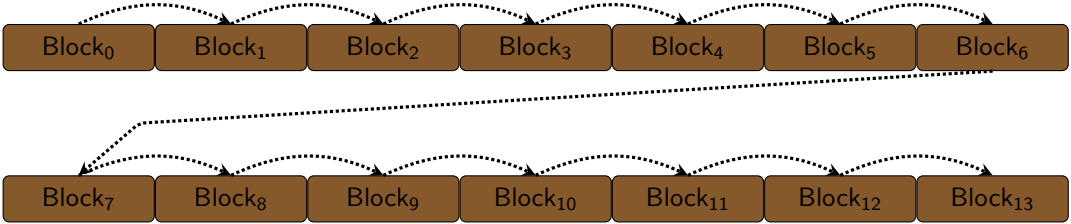
# Outline

1. STL and memory management
   - Dynamic memory allocations in C++
   - Allocation patterns
   - Typical allocation goals and strategies
2. Supporting node-allocation in the STL
   - Proposal
   - Traversal without pointer chasing
   - Reducing the pointer overhead in nodes

# Linked list travesal illustration

## Can we avoid chasing pointers?

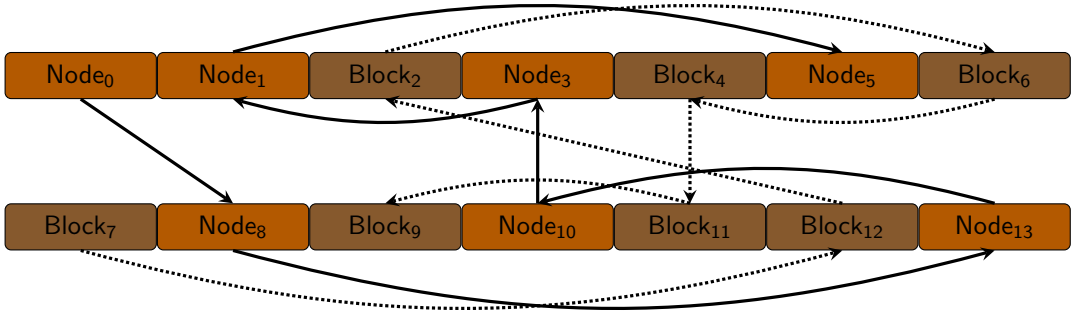
1. Initial memory state. Available memory.
2. After insertion and removal we have chaos.
3. Traversal chasing pointers jumps to slow memory.
4. Alternative: Visit blocks sequentially. Ignore unused.



# Linked list travesal illustration

## Can we avoid chasing pointers?

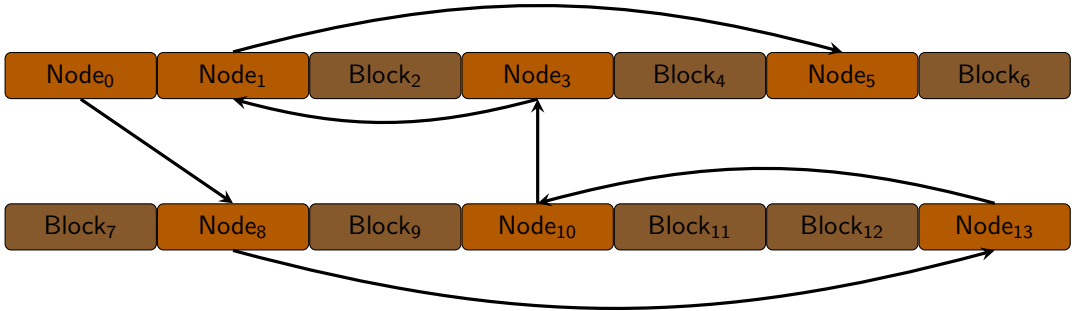
1. Initial memory state. Available memory.
2. After insertion and removal we have chaos.
3. Traversal chasing pointers jumps to slow memory.
4. Alternative: Visit blocks sequentially. Ignore unused.



# Linked list travesal illustration

## Can we avoid chasing pointers?

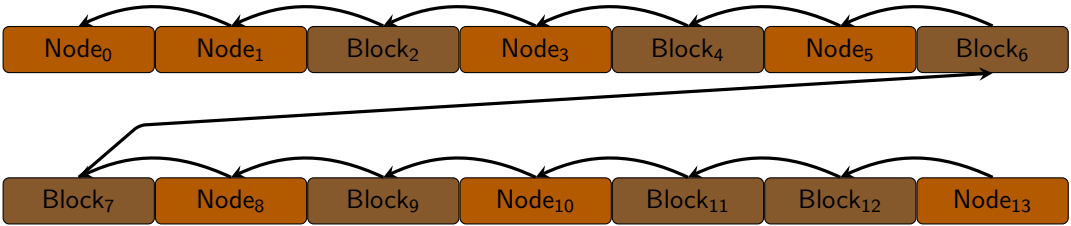
1. Initial memory state. Available memory.
2. After insertion and removal we have chaos.
3. Traversal chasing pointers jumps to slow memory.
4. Alternative: Visit blocks sequentially. Ignore unused.



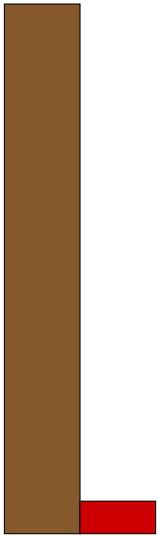
# Linked list travesal illustration

## Can we avoid chasing pointers?

1. Initial memory state. Available memory.
2. After insertion and removal we have chaos.
3. Traversal chasing pointers jumps to slow memory.
4. Alternative: Visit blocks sequentially. Ignore unused.



# Benchmark: pointer chasing vs. sequential traversal



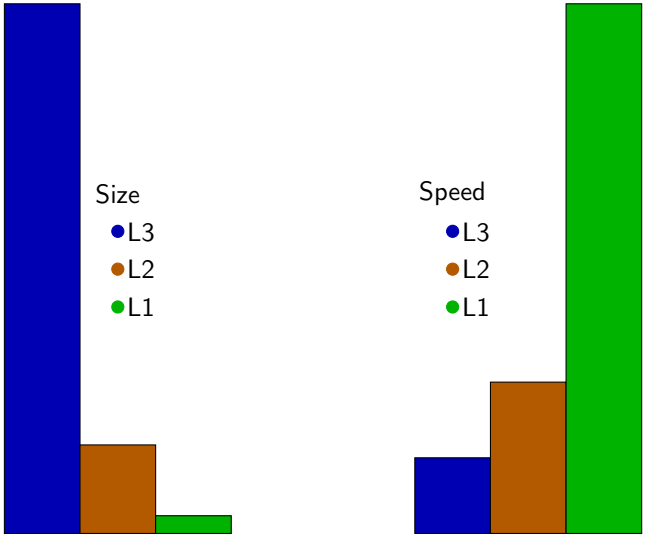
Traversal time
- Pointer chasing
- Sequential

## Benchmark

- Full allocated buffer.
- Deallocating lots of elements would eventually make both perform the same.

# Why it is so important to avoid jumping around

## Cache Hierarchy - Speed




### Intel Haswell Mobile

- Register: Fastest
- L0: 6 KiB
- L1: 128 KiB - 700 GiB/s
- L2: 1 MiB - 200 GiB/s
- L3: 6 MiB - 100 GB/s
- L4: 128 MiB - 40 GB/s
- Main memory – GiB - 10 GB/s

# Alternative to pointer chasing

## Typical example

```
// Allocator configured to chunks of 8
// nodes.
using alloc_type =
    node_allocator<int, node_type, 8>;

std::list<int, alloc_type> l {11, 23, 35};

auto strg =
  l.get_allocator().get_node_storage();

const auto n =
    std::count_if( std::begin(strg)
                 , std::end(strg),
                 , in_use());
// n = 3
```

### Code example

- Constructor marks the memory block in use.
- Destructor mark them free.
- `std::count_if` traverses the underlying buffer ignoring unused nodes.

## A Flame About 64-bit Pointers

*It is absolutely idiotic to have 64-bit pointers when I compile a program that uses less than 4 gigabytes of RAM. When such pointer values appear inside a struct, they not only waste half the memory, they effectively throw away half of the cache.* (Donald Knuth)

## A Flame About 64-bit Pointers

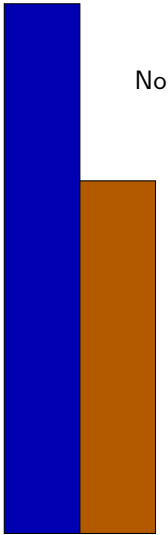
*It is absolutely idiotic to have 64-bit pointers when I compile a program that uses less than 4 gigabytes of RAM. When such pointer values appear inside a struct, they not only waste half the memory, they effectively throw away half of the cache.* (Donald Knuth)

## Let us further explore this idea

A node allocator uses a deque-like data structure whose elements have indexed access. Nodes do not need to store pointers to other nodes, they can store indexes.

# Pointers are a big overhead on nodes with small sizes

## Nodes with `std::string`



Node sizes

- pointer
- `std::uint32_t`

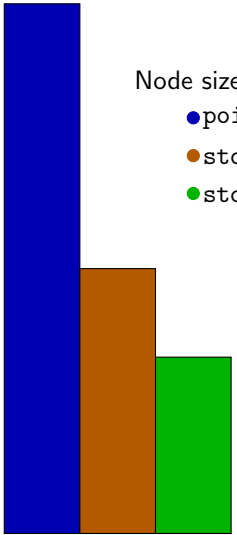
```
// Typical node
struct node {
  link_type prev;
  link_type next;
  std::string info;
};
```

### Thoughts

Reducing further to `std::uint16_t` won't make any difference since the spared space will be wasted on padding due to `std::string` alignment.

# Pointers are a big overhead on nodes with small sizes

## Node with `int`



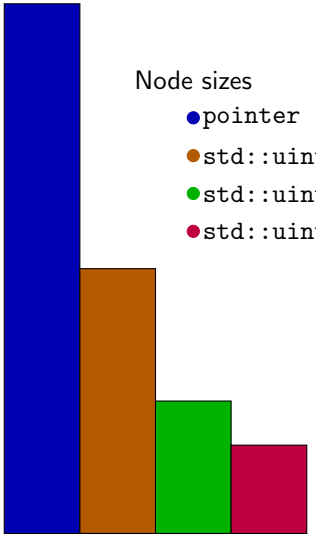

```
// Typical node
struct node {
  link_type prev;
  link_type next;
  int info;
};
```

### Thoughts

- On a 64 bits system the struct size is 24 bytes.
- Four bytes are wasted on padding.
- Using 32 bits pointers may not be an option.
- Even if it were, 16 bits are usually enough.

# Pointers are a big overhead on nodes with small sizes

## Node with `short`



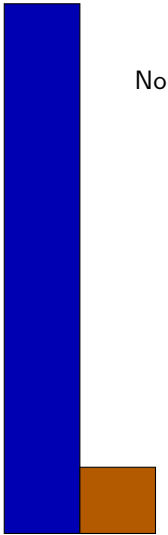

```
// Typical node. Often 16 bits integers are
// enough but not used in practice since spared
// space is wasted anyway due to padding.
struct node {
  link_type prev;
  link_type next;
  short info;
};
```

### Thoughts

Pretty useful if the range of values $[0, 65536]$ is enough.

# Pointers are a big overhead on nodes with small sizes

## Nodes sizes if indexing were possible: The extreme case

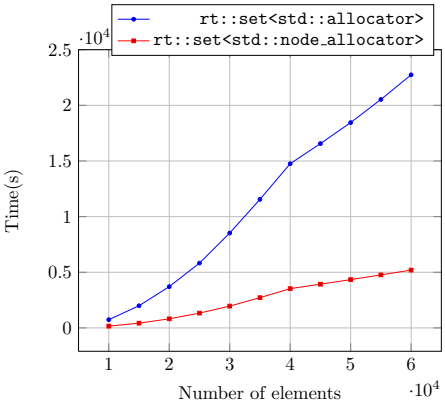


Node sizes

- pointer
- std::uint8_t

```
// Extreme case. Small value type.
// Small number of elements.
struct node {
  link_type prev;
  link_type next;
  unsigned char ages;
};
```

# Speedup due to data locality




## Why so much speed up?

- Same data structure.
- But 4 times more data in the cache.
- Uses `std::uint16_t` as index and `short` as `value_type`
- Do not forget that 16 bits is enough for $65,536$ elements.

# How to achieve this without breaking code

## Back to fancy pointers

```
template<class Alloc>
struct allocator_traits {
  // Equal to Alloc::link_type if present,
  // Alloc::pointer otherwise.
  using link_type = ...
};
```

### How it works

- Nodes store links to other nodes not pointers.
- Define some convertions from `link_type` to `pointer` and back.
- The `link_type` contains only an index (e.g. `std::uint16_t`)
- `link_type` equal to `pointer` in current cases.
- Containers rebind their nodes to the `link_type` instead of the `void_pointer`

## Summary

- Incorporate node allocation in the STL.
- Alternative to pointer chasing.
- Reduce pointers overhead in nodes.
- Make STL containers more flexible.
- Reduce the need of new containers to handle corner cases.


## Acknowledgements

- `std-proposals`
- `WG21-SG14`
- `ccppbrasil`

**Thank you! Questions?**