C++ and Memory: Between Correctness and Performance

Ulrich Drepper
Consulting Engineer, Office of the CTO
<drepper@redhat.com>

C

- Designed as a language to implement OSes
  - Full control over all resources
  - Mostly direct control over code generation
- Drawbacks
  - Everything programmer's responsibility

C++ 1998

- Inherit properties from C
- Memory handling mostly equivalent
  - Allocation **size** in most cases implicit (`new`)
- Help cleaning up objects added
  - Destructors
  - `catch` blocks

Problems

```
struct obj {
  obj* next;
  int* p1;
  int* p2;
  int* p3;
  int* p4;
  obj(obj* n = nullptr) : next(n),
    p2(new int[10]), p3(new int(4)),
    p4(new int(44)) {}
  ~obj() { delete p1; delete p2;
           delete p4; }
};
void g(int* p) { delete p; }
void f() {
  obj o;
  g(o.p4);
}
```

Problems

```
struct obj {
  obj* next;
  foo* p1;
  foo* p2;
  foo* p3;
  foo* p4;
  obj(obj* n = nullptr) : next(n),
    p2(new foo[10]), p3(new foo(4)),
    p4(new foo(44)) {}
  ~obj() { delete p1; delete p2;
           delete p4; }
};
void g(foo* p) { delete p; }
void f() {
  obj o;
  g(o.p4);
}
```

- Uninitialized pointer

Problems

```
struct obj {
  obj* next;
  foo* p1;
  foo* p2;
  foo* p3;
  foo* p4;
  obj(obj* n = nullptr) : next(n),
    p2(new foo[10]), p3(new foo(4)),
    p4(new foo(44)) {}
  ~obj() { delete p1; delete p2;
           delete p4; }
};
void g(foo* p) { delete p; }
void f() {
  obj o;
  g(o.p4);
}
```

- Uninitialized pointer
- Array allocation, scalar deletion

Problems

```
struct obj {
  obj* next;
  foo* p1;
  foo* p2;
  foo* p3;
  foo* p4;
  obj(obj* n = nullptr) : next(n),
    p2(new foo[10]), p3(new foo(4)),
    p4(new foo(44)) {}
  ~obj() { delete p1; delete p2;
           delete p4; }
};
void g(foo* p) { delete p; }
void f() {
  obj o;
  g(o.p4);
}
```

- Uninitialized pointer
- Array allocation, scalar deletion
- Missing deletion

Problems

```
struct obj {
  obj* next;
  foo* p1;
  foo* p2;
  foo* p3;
  foo* p4;
  obj(obj* n = nullptr) : next(n),
    p2(new foo[10]), p3(new foo(4)),
    p4(new foo(44)) {}
  ~obj() { delete p1; delete p2;
           delete p4; }
};
void g(foo* p) { delete p; }
void f() {
  obj o;
  g(o.p4);
}
```

- Uninitialized pointer
- Array allocation, scalar deletion
- Missing deletion
- Double deletion

Problems

```
struct obj {
  obj* next;
  foo* p1;
  foo* p2;
  foo* p3;
  foo* p4;
  obj(obj* n = nullptr) : next(n),
    p2(new foo[10]), p3(new foo(4)),
    p4(new foo(44)) {}
  ~obj() { delete p1; delete p2;
           delete p4; }
};
void g(foo* p) { delete p; }
void f() {
  obj o;
  g(o.p4);
}
```

- Uninitialized pointer
- Array allocation, scalar deletion
- Missing deletion
- Double deletion
- Potentially circular

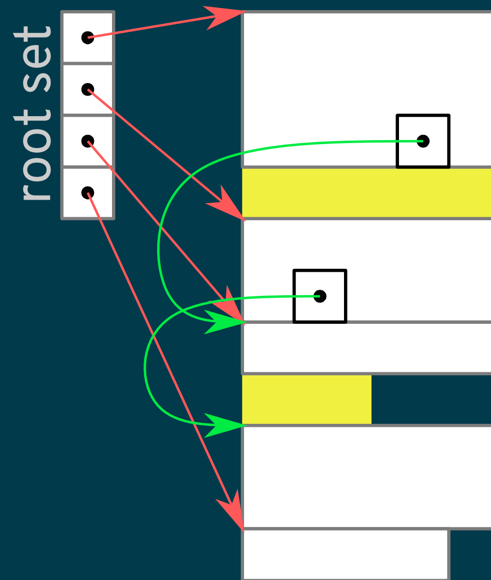
redhat.

Automation Necessary!

redhat.

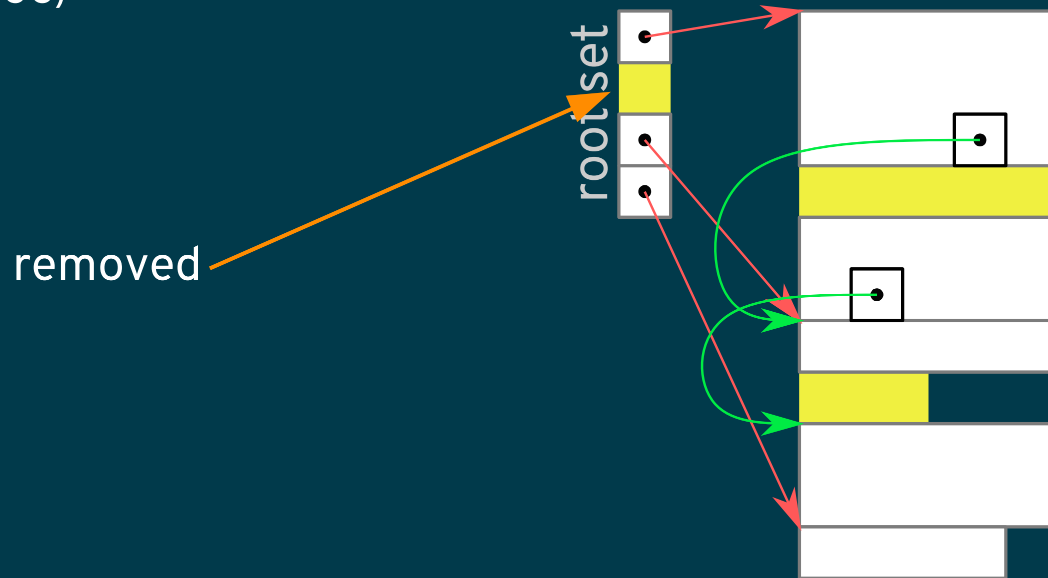
Garbage Collection

Also for C++ (Boehm GC)



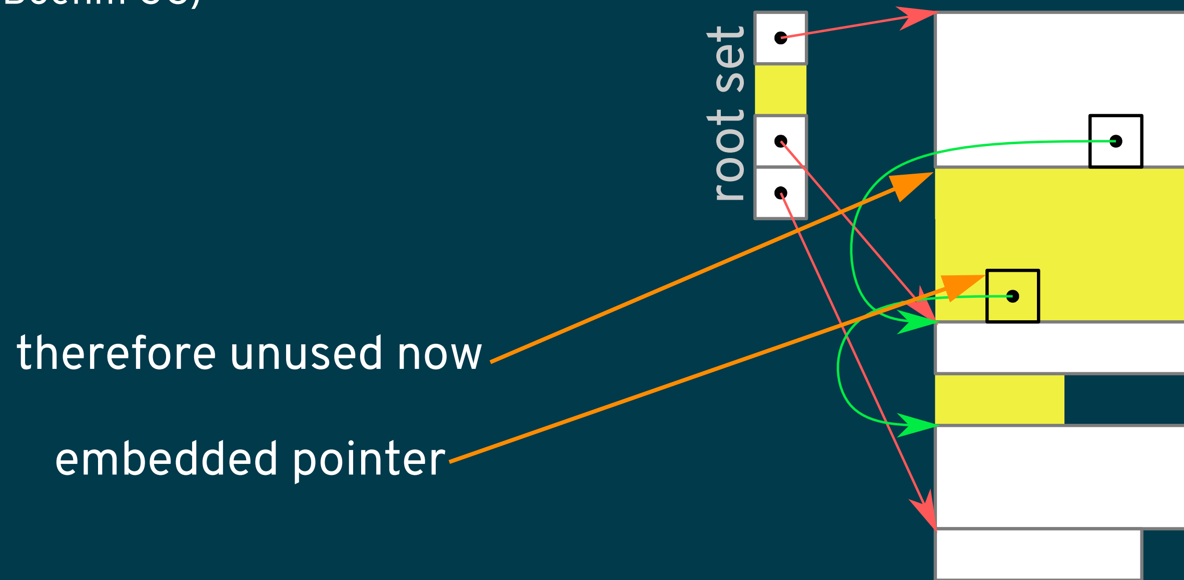
Garbage Collection

Also for C++ (Boehm GC)



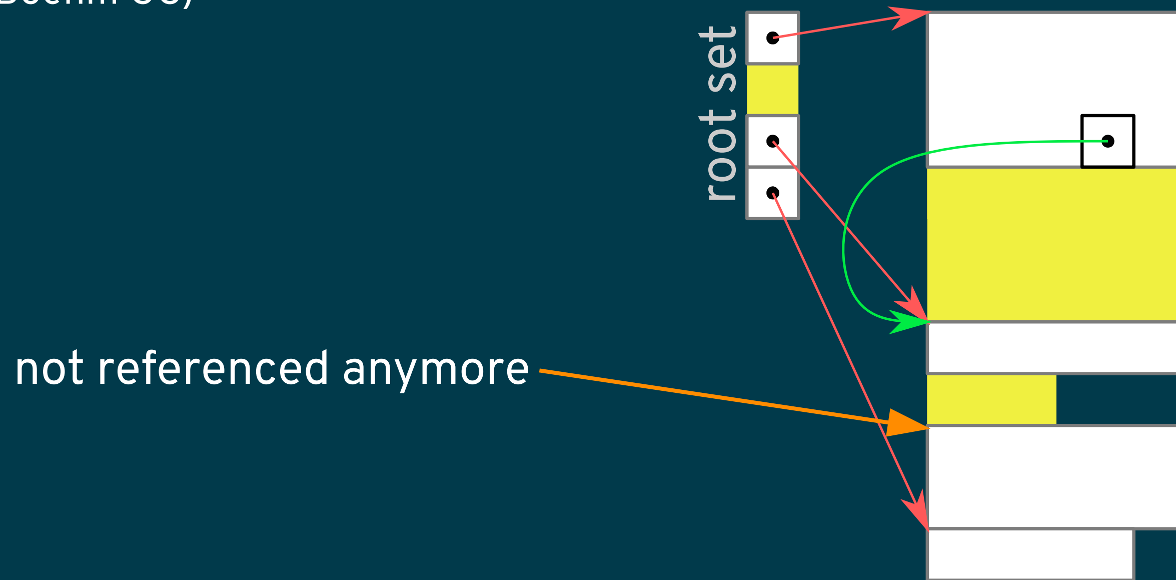
Garbage Collection

Also for C++ (Boehm GC)



Garbage Collection

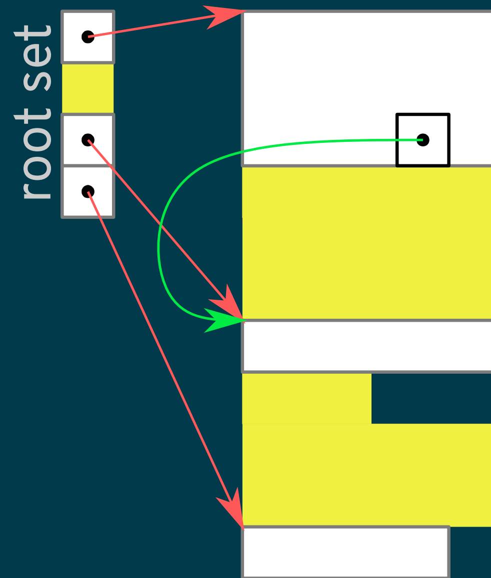
Also for C++ (Boehm GC)



Garbage Collection

Not free

- Secondary memory handler
- Periodic interruption
- Large memory footprint when tracing valid objects
- Pessimistic in absence of 100% introspection



Automatic Memory

On Stack

```
#include <array>
#include <numeric>

int rs(int s) {
  std::array<int,100> ar;
  std::iota(ar.begin(), ar.end(), s);
  return std::accumulate(ar.begin(),
                         ar.end(), 0);
}
```

```
0000000000000000 <rs(int)>:
   0:   48 81 ec 20 01 00 00    sub    $0x120,%rsp
   7:   48 8d 44 24 88          lea    -0x78(%rsp),%rax
   c:   0f 1f 40 00             nopl   0x0(%rax)
  10:   89 38                   mov    %edi,(%rax)
  12:   48 8d 8c 24 18 01 00    lea    0x118(%rsp),%rcx
  19:   00
  1a:   48 83 c0 04             add    $0x4,%rax
  1e:   83 c7 01                add    $0x1,%edi
  21:   48 39 c8                cmp    %rcx,%rax
  24:   75 ea                   jne    10 <rs(int)+0x10>
  26:   48 8d 54 24 88          lea    -0x78(%rsp),%rdx
  2b:   31 c0                   xor    %eax,%eax
  2d:   0f 1f 00                nopl   (%rax)
  30:   03 02                   add    (%rdx),%eax
  32:   48 8d b4 24 18 01 00    lea    0x118(%rsp),%rsi
  39:   00
  3a:   48 83 c2 04             add    $0x4,%rdx
  3e:   48 39 f2                cmp    %rsi,%rdx
  41:   75 ed                   jne    30 <rs(int)+0x30>
  43:   48 81 c4 20 01 00 00    add    $0x120,%rsp
  4a:   c3                      retq
```

On Stack

```cpp
#include <array>
#include <numeric>

int rs(int s) {
  std::array<int,100> ar;
  std::iota(ar.begin(), ar.end(), s);
  return std::accumulate(ar.begin(),
                         ar.end(), 0);
}
```

```
0000000000000000 <rs(int)>:
   0:  48 81 ec 20 01 00 00     sub     $0x120,%rsp
   7:  48 8d 44 24 88           lea     -0x78(%rsp),%rax
   c:  0f 1f 40 00              nopl    0x0(%rax)
  10:  89 38                    mov     %edi,(%rax)
  12:  48 8d 8c 24 18 01 00     lea     0x118(%rsp),%rcx
  19:  00
  1a:  48 83 c0 04              add     $0x4,%rax
  1e:  83 c7 01                 add     $0x1,%edi
  21:  48 39 c8                 cmp     %rcx,%rax
  24:  75 ea                    jne     10 <rs(int)+0x10>
  26:  48 8d 54 24 88           lea     -0x78(%rsp),%rdx
  2b:  31 c0                    xor     %eax,%eax
  2d:  0f 1f 00                 nopl    (%rax)
  30:  03 02                    add     (%rdx),%eax
  32:  48 8d b4 24 18 01 00     lea     0x118(%rsp),%rsi
  39:  00
  3a:  48 83 c2 04              add     $0x4,%rdx
  3e:  48 39 f2                 cmp     %rsi,%rdx
  41:  75 ed                    jne     30 <rs(int)+0x30>
  43:  48 81 c4 20 01 00 00     add     $0x120,%rsp
  4a:  c3                       retq
```

On Stack

```
#include <array>
#include <numeric>

int rs(int s) {
  std::array<int,100> ar;
  std::iota(ar.begin(), ar.end(), s);
  return std::accumulate(ar.begin(),
                         ar.end(), 0);
}
```

```
0000000000000000 <rs(int)>:
   0:  48 81 ec 20 01 00 00     sub    $0x120,%rsp
   7:  48 8d 44 24 88           lea    -0x78(%rsp),%rax
   c:  0f 1f 40 00              nopl   0x0(%rax)
  10:  89 38                    mov    %edi,(%rax)
  12:  48 8d 8c 24 18 01 00     lea    0x118(%rsp),%rcx
  19:  00
  1a:  48 83 c0 04              add    $0x4,%rax
  1e:  83 c7 01                 add    $0x1,%edi
  21:  48 39 c8                 cmp    %rcx,%rax
  24:  75 ea                    jne    10 <rs(int)+0x10>
  26:  48 8d 54 24 88           lea    -0x78(%rsp),%rdx
  2b:  31 c0                    xor    %eax,%eax
  2d:  0f 1f 00                 nopl   (%rax)
  30:  03 02                    add    (%rdx),%eax
  32:  48 8d b4 24 18 01 00     lea    0x118(%rsp),%rsi
  39:  00
  3a:  48 83 c2 04              add    $0x4,%rdx
  3e:  48 39 f2                 cmp    %rsi,%rdx
  41:  75 ed                    jne    30 <rs(int)+0x30>
  43:  48 81 c4 20 01 00 00     add    $0x120,%rsp
  4a:  c3                       retq
```

Dynamic

- Stack size
  - Limited at runtime
  - Static size without check at compile time
- Variable length arrays dangerous!
  - No standard support
    - At most `https://wg21.link/P0785R0`

Not the Same!

```cpp
#include <array>
#include <numeric>

int rs(int s) {
  std::array<int,100> ar;
  std::iota(ar.begin(), ar.end(), s);
  return std::accumulate(ar.begin(),
                         ar.end(), 0);
}
```

```cpp
#include <vector>
#include <numeric>

int rs(int s) {
  std::vector<int> ar(100);
  std::iota(ar.begin(), ar.end(), s);
  return std::accumulate(ar.begin(),
                         ar.end(), 0);
}
```

Not the Same!

```cpp
#include <array>
#include <numeric>

int rs(int s) {
  std::array<int,100> ar;
  std::iota(ar.begin(), ar.end(), s);
  return std::accumulate(ar.begin(),
                         ar.end(), 0);
}
```

```cpp
#include <vector>
#include <numeric>

int rs(int s) {
  std::vector<int> ar(100);
  std::iota(ar.begin(), ar.end(), s);
  return std::accumulate(ar.begin(),
                         ar.end(), 0);
}
```

BUT HOW DIFFERENT?

How Different?

```
0000000000000000 <rs(int)>:
   0:     sub    $0x120,%rsp
   7:     lea    -0x78(%rsp),%rax
   c:     nopl   0x0(%rax)
  10:     mov    %edi,(%rax)
  12:     lea    0x118(%rsp),%rcx
  1a:     add    $0x4,%rax
  1e:     add    $0x1,%edi
  21:     cmp    %rcx,%rax
  24:     jne    10 <rs(int)+0x10>
  26:     lea    -0x78(%rsp),%rdx
  2b:     xor    %eax,%eax
  2d:     nopl   (%rax)
  30:     add    (%rdx),%eax
  32:     lea    0x118(%rsp),%rsi
  3a:     add    $0x4,%rdx
  3e:     cmp    %rsi,%rdx
  41:     jne    30 <rs(int)+0x30>
  43:     add    $0x120,%rsp
  4a:     retq
```

```
0000000000000000 <rs(int)>:
   0:     push   %rbx
   1:     mov    %edi,%ebx
   3:     mov    $0x190,%edi
   8:     callq  operator new(unsigned long)
   d:     lea    0x190(%rax),%rsi
  14:     mov    %rax,%rdx
  17:     nopw   0x0(%rax,%rax,1)
  20:     movl   $0x0,(%rdx)
  26:     add    $0x4,%rdx
  2a:     cmp    %rdx,%rsi
  2d:     jne    20 <rs(int)+0x20>
  2f:     mov    %rax,%rdx
  32:     lea    0x64(%rbx),%r8d
  36:     mov    %rax,%rcx
  39:     nopl   0x0(%rax)
  40:     mov    %ebx,(%rcx)
  42:     add    $0x1,%ebx
  45:     add    $0x4,%rcx
  49:     cmp    %r8d,%ebx
  4c:     jne    40 <rs(int)+0x40>
  4e:     xor    %ebx,%ebx
  50:     add    (%rdx),%ebx
  52:     add    $0x4,%rdx
  56:     cmp    %rsi,%rdx
  59:     jne    50 <rs(int)+0x50>
  5b:     mov    %rax,%rdi
  5e:     callq  operator delete(void*)
  63:     mov    %ebx,%eax
  65:     pop    %rbx
  66:     retq
```

How Different?

```
0000000000000000 <rs(int)>:
   0:     sub     $0x120,%rsp
   7:     lea     -0x78(%rsp),%rax
   c:     nopl    0x0(%rax)
  10:     mov     %edi,(%rax)
  12:     lea     0x118(%rsp),%rcx
  1a:     add     $0x4,%rax
  1e:     add     $0x1,%edi
  21:     cmp     %rcx,%rax
  24:     jne     10 <rs(int)+0x10>
  26:     lea     -0x78(%rsp),%rdx
  2b:     xor     %eax,%eax
  2d:     nopl    (%rax)
  30:     add     (%rdx),%eax
  32:     lea     0x118(%rsp),%rsi
  3a:     add     $0x4,%rdx
  3e:     cmp     %rsi,%rdx
  41:     jne     30 <rs(int)+0x30>
  43:     add     $0x120,%rsp
  4a:     retq
```

std::iota

std::accumulate

```
0000000000000000 <rs(int)>:
   0:     push    %rbx
   1:     mov     %edi,%ebx
   3:     mov     $0x190,%edi
   8:     callq   operator new(unsigned long)
   d:     lea     0x190(%rax),%rsi
  14:     mov     %rax,%rdx
  17:     nopw    0x0(%rax,%rax,1)
  20:     movl    $0x0,(%rdx)
  26:     add     $0x4,%rdx
  2a:     cmp     %rdx,%rsi
  2d:     jne     20 <rs(int)+0x20>
  2f:     mov     %rax,%rdx
  32:     lea     0x64(%rbx),%r8d
  36:     mov     %rax,%rcx
  39:     nopl    0x0(%rax)
  40:     mov     %ebx,(%rcx)
  42:     add     $0x1,%ebx
  45:     add     $0x4,%rcx
  49:     cmp     %r8d,%ebx
  4c:     jne     40 <rs(int)+0x40>
  4e:     xor     %ebx,%ebx
  50:     add     (%rdx),%ebx
  52:     add     $0x4,%rdx
  56:     cmp     %rsi,%rdx
  59:     jne     50 <rs(int)+0x50>
  5b:     mov     %rax,%rdi
  5e:     callq   operator delete(void*)
  63:     mov     %ebx,%eax
  65:     pop     %rbx
  66:     retq
```

How Different?

```
0000000000000000 <rs(int)>:
   0:     sub     $0x120,%rsp
   7:     lea     -0x78(%rsp),%rax
   c:     nopl    0x0(%rax)
  10:     mov     %edi,(%rax)
  12:     lea     0x118(%rsp),%rcx
  1a:     add     $0x4,%rax
  1e:     add     $0x1,%edi
  21:     cmp     %rcx,%rax
  24:     jne     10 <rs(int)+0x10>
  26:     lea     -0x78(%rsp),%rdx
  2b:     xor     %eax,%eax
  2d:     nopl    (%rax)
  30:     add     (%rdx),%eax
  32:     lea     0x118(%rsp),%rsi
  3a:     add     $0x4,%rdx
  3e:     cmp     %rsi,%rdx
  41:     jne     30 <rs(int)+0x30>
  43:     add     $0x120,%rsp
  4a:     retq
```

```
0000000000000000 <rs(int)>:
   0:     push    %rbx
   1:     mov     %edi,%ebx
   3:     mov     $0x190,%edi
   8:     callq   operator new(unsigned long)
   d:     lea     0x190(%rax),%rsi
  14:     mov     %rax,%rdx
  17:     nopw    0x0(%rax,%rax,1)
  20:     movl    $0x0,(%rdx)
  26:     add     $0x4,%rdx
  2a:     cmp     %rdx,%rsi
  2d:     jne     20 <rs(int)+0x20>
  2f:     mov     %rax,%rdx
  32:     lea     0x64(%rbx),%r8d
  36:     mov     %rax,%rcx
  39:     nopl    0x0(%rax)
  40:     mov     %ebx,(%rcx)
  42:     add     $0x1,%ebx
  45:     add     $0x4,%rcx
  49:     cmp     %r8d,%ebx
  4c:     jne     40 <rs(int)+0x40>
  4e:     xor     %ebx,%ebx
  50:     add     (%rdx),%ebx
  52:     add     $0x4,%rdx
  56:     cmp     %rsi,%rdx
  59:     jne     50 <rs(int)+0x50>
  5b:     mov     %rax,%rdi
  5e:     callq   operator delete(void*)
  63:     mov     %ebx,%eax
  65:     pop     %rbx
  66:     retq
```

??? (lines 14–2d of right listing)

std::iota (highlighted: left 10–24, right 40–4c)

std::accumulate (highlighted: left 30–41, right 50–59)

redhat.

How Different?

```
0000000000000000 <rs(int)>:
   0:     sub     $0x120,%rsp
   7:     lea     -0x78(%rsp),%rax
   c:     nopl    0x0(%rax)
  10:     mov     %edi,(%rax)
  12:     lea     0x118(%rsp),%rcx
  1a:     add     $0x4,%rax
  1e:     add     $0x1,%edi
  21:     cmp     %rcx,%rax
  24:     jne     10 <rs(int)+0x10>
  26:     lea     -0x78(%rsp),%rdx
  2b:     xor     %eax,%eax
  2d:     nopl    (%rax)
  30:     add     (%rdx),%eax
  32:     lea     0x118(%rsp),%rsi
  3a:     add     $0x4,%rdx
  3e:     cmp     %rsi,%rdx
  41:     jne     30 <rs(int)+0x30>
  43:     add     $0x120,%rsp
  4a:     retq
```

23.3.6.2 vector constructors, copy, and assignment
Constructs a vector with n *default-inserted elements using the specified allocator.*

std::iota

std::accumulate

```
0000000000000000 <rs(int)>:
   0:     push    %rbx
   1:     mov     %edi,%ebx
   3:     mov     $0x190,%edi
   8:     callq   operator new(unsigned long)
   d:     lea     0x190(%rax),%rsi
  14:     mov     %rax,%rdx
  17:     nopw    0x0(%rax,%rax,1)
  20:     movl    $0x0,(%rdx)
  26:     add     $0x4,%rdx
  2a:     cmp     %rdx,%rsi
  2d:     jne     20 <rs(int)+0x20>
  2f:     mov     %rax,%rdx
  32:     lea     0x64(%rbx),%r8d
  36:     mov     %rax,%rcx
  39:     nopl    0x0(%rax)
  40:     mov     %ebx,(%rcx)
  42:     add     $0x1,%ebx
  45:     add     $0x4,%rcx
  49:     cmp     %r8d,%ebx
  4c:     jne     40 <rs(int)+0x40>
  4e:     xor     %ebx,%ebx
  50:     add     (%rdx),%ebx
  52:     add     $0x4,%rdx
  56:     cmp     %rsi,%rdx
  59:     jne     50 <rs(int)+0x50>
  5b:     mov     %rax,%rdi
  5e:     callq   operator delete(void*)
  63:     mov     %ebx,%eax
  65:     pop     %rbx
  66:     retq
```

redhat.

Keeping vector

```diff
--- various/vectoronstack.cc 2018-10-21 07:33:34.694472765 +0200
+++ various/vectoronstack2.cc 2018-10-21 20:45:26.017628366 +0200
@@ -1,9 +1,11 @@
 #include <vector>
 #include <numeric>
+#include <algorithm>

 int rs(int s) {
-  std::vector<int> ar(100);
-  std::iota(ar.begin(), ar.end(), s);
+  std::vector<int> ar;
+  ar.reserve(100);
+  std::generate_n(std::back_inserter(ar), 100, [&s](){ return s++; });
   return std::accumulate(ar.begin(), ar.end(), 0);
 }
```

Keeping vector

```
--- various/vectoronstack.cc 2018-10-21 07:33:34.694472765 +0200
+++ various/vectoronstack2.cc 2018-10-21 20:45:26.017628366 +0200
@@ -1,9 +1,11 @@
 #include <vector>
 #include <numeric>
+#include <algorithm>

 int rs(int s) {
-  std::vector<int> ar(100);
-  std::iota(ar.begin(), ar.end(), s);
+  std::vector<int> ar;
+  ar.reserve(100);
+  std::generate_n(std::back_inserter(ar), 100, [&s](){ return s++; });
   return std::accumulate(ar.begin(), ar.end(), 0);
 }
```

Generates a whole lot of code...

redhat.

Our Own dynamic array

```
--- various/arrayonstack.cc   2018-10-21 07:33:47.134426871 +0200
+++ various/arrayonstack2.cc  2018-10-22 13:11:37.122031118 +0200
@@ -1,8 +1,20 @@
 #include <array>
 #include <numeric>

+template<typename T, size_t N>
+struct dynarray {
+  typedef std::array<T,N> array_type;
+  array_type* m;
+  dynarray() { m = new array_type; }
+  ~dynarray() { delete m; }
+  T& operator[](size_t n) { return m[n]; }
+  T operator[](size_t n) const { return m[n]; }
+  typename array_type::iterator begin() { return m->begin(); }
+  typename array_type::iterator end() { return m->end(); }
+};
+
 int rs(int s) {
-  std::array<int,100> ar;
+  dynarray<int,100> ar;
   std::iota(ar.begin(), ar.end(), s);
   return std::accumulate(ar.begin(), ar.end(), 0);
 }
```

How Different?

```
0000000000000000 <rs(int)>:
   0:	sub    $0x120,%rsp
   7:	lea    -0x78(%rsp),%rax
   c:	nopl   0x0(%rax)
  10:	mov    %edi,(%rax)
  12:	lea    0x118(%rsp),%rcx
  1a:	add    $0x4,%rax
  1e:	add    $0x1,%edi
  21:	cmp    %rcx,%rax
  24:	jne    10 <rs(int)+0x10>
  26:	lea    -0x78(%rsp),%rdx
  2b:	xor    %eax,%eax
  2d:	nopl   (%rax)
  30:	add    (%rdx),%eax
  32:	lea    0x118(%rsp),%rsi
  3a:	add    $0x4,%rdx
  3e:	cmp    %rsi,%rdx
  41:	jne    30 <rs(int)+0x30>
  43:	add    $0x120,%rsp
  4a:	retq
```

std::iota

std::accumulate

```
0000000000000000 <rs(int)>:
   0:	push   %rbx
   1:	mov    %edi,%ebx
   3:	mov    $0x190,%edi
   8:	callq  operator new(unsigned long)
   d:	mov    %rax,%rdx
  10:	lea    0x190(%rax),%rsi
  17:	mov    %rax,%rcx
  1a:	nopw   0x0(%rax,%rax,1)
  20:	mov    %ebx,(%rcx)
  22:	add    $0x4,%rcx
  26:	add    $0x1,%ebx
  29:	cmp    %rsi,%rcx
  2c:	jne    20 <rs(int)+0x20>
  2e:	xor    %ebx,%ebx
  30:	add    (%rdx),%ebx
  32:	add    $0x4,%rdx
  36:	cmp    %rsi,%rdx
  39:	jne    30 <rs(int)+0x30>
  3b:	mov    %rax,%rdi
  3e:	mov    $0x190,%esi
  43:	callq  operator delete(void*, unsigned long)
  48:	mov    %ebx,%eax
  4a:	pop    %rbx
  4b:	retq
```

redhat.

Reference Counting

redhat.

std::shared_ptr – easy to use

```
#include <memory>

typedef std::shared_ptr<int> ptr_type;

ptr_type f(int a) {
  return std::make_shared<int>(a);
}

int g(int a, int b) {
  auto p = f(a);
  return *p + b;
}

int ga, gb;

int main() {
  return g(ga, gb);
}
```

std::shared_ptr – easy to use but expensive

```cpp
#include <memory>

using ptr_type = std::shared_ptr<int>;

ptr_type f(int a) {
  return std::make_shared<int>(a);
}

int g(int a, int b) {
  auto p = f(a);
  return *p + b;
}

int ga, gb;

int main() {
  return g(ga, gb);
}
```

```
0000000000400790 <g(int, int)>:
  400790:  push   %r12
  400792:  mov    %esi,%r12d
  400795:  push   %rbp
  400796:  mov    %edi,%ebp
  400798:  mov    $0x18,%edi
  40079d:  push   %rbx
  40079e:  callq  operator new(unsigned long)
  4007a3:  mov    %rax,%rbx
  4007a6:  movabs $0x100000001,%rax
  4007b0:  mov    %ebp,0x10(%rbx)
  4007b3:  add    %r12d,%ebp
  4007b6:  mov    $__pthread_key_create,%r12d
  4007bc:  mov    %rax,0x8(%rbx)
  4007c0:  movq   $0x400a70,(%rbx)
  4007c7:  test   %r12,%r12
  4007ca:  je     4007e0
  4007cc:  lock subl $0x1,0x8(%rbx)
  4007d1:  je     400820 <g(int, int)+0x90>
  4007d3:  mov    %ebp,%eax
  4007d5:  pop    %rbx
  4007d6:  pop    %rbp
  4007d7:  pop    %r12
  4007d9:  retq
  4007e0:  movl   $0x0,0x8(%rbx)
  4007e7:  mov    $std::shared_ptr<int>::_M_dispose,%eax
  4007ec:  mov    %rbx,%rdi
  4007ef:  callq  *%rax
  4007f1:  test   %r12,%r12
  4007f4:  je     400810
  4007f6:  mov    $0xffffffff,%eax
  4007fb:  lock xadd %eax,0xc(%rbx)
  400800:  cmp    $0x1,%eax
  400803:  jne    4007d3
  400805:  mov    (%rbx),%rax
  400808:  mov    %rbx,%rdi
  40080b:  callq  *0x18(%rax)
  40080e:  jmp    4007d3
  400810:  mov    0xc(%rbx),%eax
  400813:  lea    -0x1(%rax),%edx
  400816:  mov    %edx,0xc(%rbx)
  400819:  jmp    400800
  400820:  mov    (%rbx),%rax
  400823:  mov    0x10(%rax),%rax
  400827:  jmp    4007ec
```

f(a)

Threads?

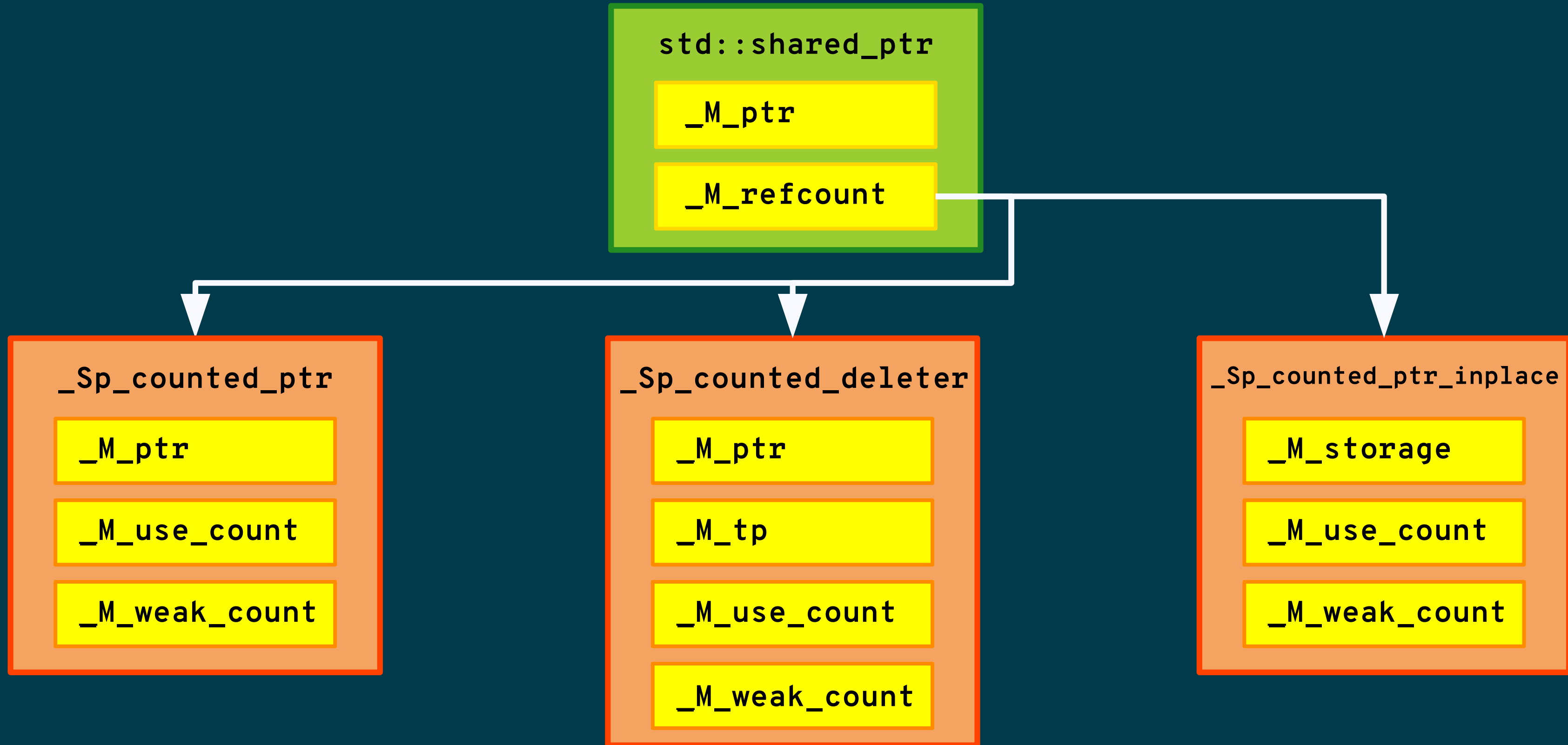
No Threads!

Threads?

No Threads!

redhat.

std::shared_ptr Implementation



Passing `std::shared_ptr`

```cpp
#include <memory>
using ptr_type = std::shared_ptr<int>;
ptr_type pi;

int f1(ptr_type a, int b) {
  return *a + b;
}
int c1(int b) {
  return f1(pi, b);
}
```

```cpp
#include <memory>
using ptr_type = std::shared_ptr<int>;
ptr_type pi;

int f2(ptr_type& a, int b) {
  return *a + b;
}
int c2(int b) {
  return f2(pi, b);
}
```

```cpp
#include <memory>
using ptr_type = std::shared_ptr<int>;
ptr_type pi;

int f3(typename ptr_type::element_type* a, int b) {
  return *a + b;
}
int c3(int b) {
  return f3(pi.get(), b);
}
```

Passing std::shared_ptr

```
f1(std::shared_ptr<int>, int):
   0:   mov     (%rdi),%rax
   3:   add     (%rax),%esi
   5:   mov     %esi,%eax
   7:   retq
```

```
f2(std::shared_ptr<int>&, int):
   0:   mov     (%rdi),%rax
   3:   add     (%rax),%esi
   5:   mov     %esi,%eax
   7:   retq
```

```
f3(int*, int):
   0:   mov     (%rdi),%eax
   2:   add     %esi,%eax
   4:   retq
```

Passing std::shared_ptr

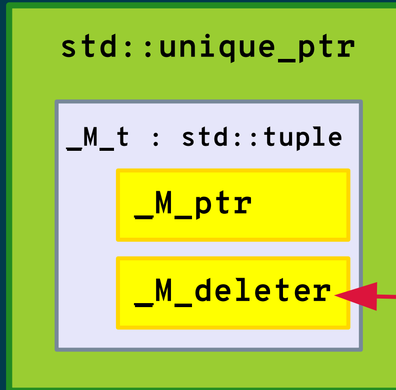
```
c1(int):
  40: push   %rbp
  41: push   %rbx
  42: sub    $0x28,%rsp
  46: mov    pi+8(%rip),%rbx
  4d: mov    pi(%rip),%rax
  54: mov    %rax,0x10(%rsp)
  59: mov    %rbx,0x18(%rsp)
  5e: test   %rbx,%rbx
  ...
```

```
c2(int):
  40: mov    %edi,%esi
  42: mov    $pi,%edi
  47: jmpq   f2(std::shared_ptr<int>&, int)
```

```
c3(int)>:
  40: mov    %edi,%esi
  42: mov    pi(%rip),%rdi
  49: jmpq   f3(int*, int)
```

Not Sharing

std::unique_ptr Implementation



Zero-sized if trivial

No Naked Pointers

```
#include <memory>

using ptr_type = std::unique_ptr<int>;

int f(ptr_type a, int b) {
  return *a + b;
}

int g(int b) {
  auto a = std::make_unique<int>(42);
  return f(a, b);
}

int h2(ptr_type a, int b) {
  return f(a, b);
}
```

No Naked Pointers

```
#include <memory>

using ptr_type = std::unique_ptr<int>;

int f(ptr_type a, int b) {
  return *a + b;
}

int g(int b) {
  auto a = std::make_unique<int>(42);
  return f(a, b);
}

int h(ptr_type a, int b) {
  return f(a, b);
}
```

Compile-time error

No Naked Pointers (corrected)

```
#include <memory>

using ptr_type = std::unique_ptr<int>;

int f(ptr_type a, int b) {
  return *a + b;
}

int g(int b) {
  auto a = std::make_unique<int>(42);
  return f(std::move(a), b);
}

int h(ptr_type a, int b) {
  return f(std::move(a), b);
}
```

No Naked Pointers (corrected)

```cpp
#include <memory>

using ptr_type = std::unique_ptr<int>;

int f(ptr_type a, int b) {
  return *a + b;
}

int g(int b) {
  auto a = std::make_unique<int>(42);
  return f(std::move(a), b);
}

int h(ptr_type a, int b) {
  return f(std::move(a), b);
}
```

```
f(std::unique_ptr<int,std::default_delete<int>>,int):
   0: mov    (%rdi),%rax
   3: add    (%rax),%esi
   5: mov    %esi,%eax
   7: retq
```

No Naked Pointers (corrected)

```
#include <memory>

using ptr_type = std::unique_ptr<int>;

int f(ptr_type a, int b) {
  return *a + b;
}

int g(int b) {
  auto a = std::make_unique<int>(42);
  return f(std::move(a), b);
}

int h(ptr_type a, int b) {
  return f(std::move(a), b);
}
```

```
g(int):
  10: push   %rbx
  11: mov    %edi,%ebx
  13: mov    $0x4,%edi
  18: sub    $0x10,%rsp
  1c: callq  operator new(unsigned long)
  21: mov    %ebx,%esi
  23: lea    0x8(%rsp),%rdi                 f(std::move(a),b)
  28: movl   $0x2a,(%rax)
  2e: mov    %rax,0x8(%rsp)
  33: callq  f(std::unique_ptr<int,std::default_delete<int>>,int)
  38: mov    0x8(%rsp),%rdi
  3d: mov    %eax,%ebx
  3f: test   %rdi,%rdi
  42: je     4e
  44: mov    $0x4,%esi
  49: callq  operator delete(void*,unsigned long)
  4e: add    $0x10,%rsp
  52: mov    %ebx,%eax
  54: pop    %rbx
  55: retq
```

No Naked Pointers (corrected)

```cpp
#include <memory>

using ptr_type = std::unique_ptr<int>;

int f(ptr_type a, int b) {
  return *a + b;
}

int g(int b) {
  auto a = std::make_unique<int>(42);
  return f(std::move(a), b);
}

int h(ptr_type a, int b) {
  return f(std::move(a), b);
}
```

```
h(std::unique_ptr<int,std::default_delete<int>>,int):
  60: push   %rbx
  61: sub    $0x10,%rsp
  65: mov    (%rdi),%rax
  68: movq   $0x0,(%rdi)
  6f: lea    0x8(%rsp),%rdi                    f(std::move(a),b)
  74: mov    %rax,0x8(%rsp)
  79: callq  f(std::unique_ptr<int,std::default_delete<int>>,int)
  7e: mov    0x8(%rsp),%rdi
  83: mov    %eax,%ebx
  85: test   %rdi,%rdi
  88: je     94
  8a: mov    $0x4,%esi
  8f: callq  operator delete(void*, unsigned long)
  94: add    $0x10,%rsp
  98: mov    %ebx,%eax
  9a: pop    %rbx
  9b: retq
```

With Naked Pointers

```cpp
#include <memory>

using ptr_type = std::unique_ptr<int>;

int f(typename ptr_type::element_type* a, int b) {
  return *a + b;
}

int g(int b) {
  auto a = 42;
  return f(&a, b);
}

int h(ptr_type a, int b) {
  return f(a.get(), b);
}
```

```
f(int*, int):
   0: mov    (%rdi),%eax
   2: add    %esi,%eax
   4: retq

g(int):
  10: sub    $0x10,%rsp
  14: mov    %edi,%esi
  16: lea    0xc(%rsp),%rdi
  1b: movl   $0x2a,0xc(%rsp)
  23: callq  f(int*, int)
  28: add    $0x10,%rsp
  2c: retq

h(std::unique_ptr<int, std::default_delete<int> >, int):
  30: mov    (%rdi),%rdi
  33: jmp    f(int*, int)
```

rvalue Reference

rvalue Reference

```
#include <memory>

using ptr_type = std::unique_ptr<int>;

ptr_type gp;

int f(ptr_type&& a, int b) {
  gp = std::move(a);
  return *gp + b;
}

int g(int b) {
  auto a = std::make_unique<int>(42);
  return f(std::move(a), b);
}

int h(ptr_type&& a, int b) {
  return f(std::move(a), b);
}
```

```
h(std::unique_ptr<int, std::default_delete<int> >&&, int)>:
  90: jmpq   f(std::unique_ptr<int,std::default_delete<int>>&&,int)
```

rvalue Reference

```
#include <memory>

using ptr_type = std::unique_ptr<int>;

ptr_type gp;

int f(ptr_type&& a, int b) {
  gp = std::move(a);
  return *gp + b;
}

int g(int b) {
  auto a = std::make_unique<int>(42);
  return f(std::move(a), b);
}

int h(ptr_type&& a, int b) {
  return f(std::move(a), b);
}
```

```
f(std::unique_ptr<int,std::default_delete<int>>&&,int):
   0: push   %rbx
   1: mov    (%rdi),%rax
   4: mov    %esi,%ebx
   6: movq   $0x0,(%rdi)
   d: mov    gp(%rip),%rdi
  14: mov    %rax,gp(%rip)
  1b: test   %rdi,%rdi
  1e: je     31
  20: mov    $0x4,%esi
  25: callq  operator delete(void*,unsigned long)
  2a: mov    gp(%rip),%rax
  31: add    (%rax),%ebx
  33: mov    %ebx,%eax
  35: pop    %rbx
  36: retq
```

rvalue Reference

```cpp
#include <memory>

using ptr_type = std::unique_ptr<int>;

ptr_type gp;

int f(ptr_type&& a, int b) {
  gp = std::move(a);
  return *gp + b;
}

int g(int b) {
  auto a = std::make_unique<int>(42);
  return f(std::move(a), b);
}

int h(ptr_type&& a, int b) {
  return f(std::move(a), b);
}
```

```
g(int):
  40: push   %rbx
  41: mov    %edi,%ebx
  43: mov    $0x4,%edi
  48: sub    $0x10,%rsp
  4c: callq   operator new(unsigned long)
  51: mov    %ebx,%esi
  53: lea    0x8(%rsp),%rdi
  58: movl   $0x2a,(%rax)
  5e: mov    %rax,0x8(%rsp)
  63: callq  f(std::unique_ptr<int,std::default_delete<int>>&&,int)
  68: mov    0x8(%rsp),%rdi
  6d: mov    %eax,%ebx
  6f: test   %rdi,%rdi
  72: je     7e
  74: mov    $0x4,%esi
  79: callq  operator delete(void*,unsigned long)
  7e: add    $0x10,%rsp
  82: mov    %ebx,%eax
  84: pop    %rbx
  85: retq
```

redhat.

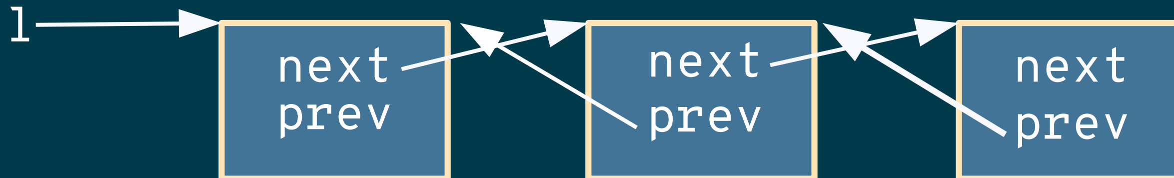
Allocators

Available in the Container Data Types

```
template <class T, class Allocator = std::allocator<T>>
std::vector {
  vector() noexcept(noexcept(Allocator()));
  explicit vector(const Allocator& alloc) noexcept;
  …
};
```

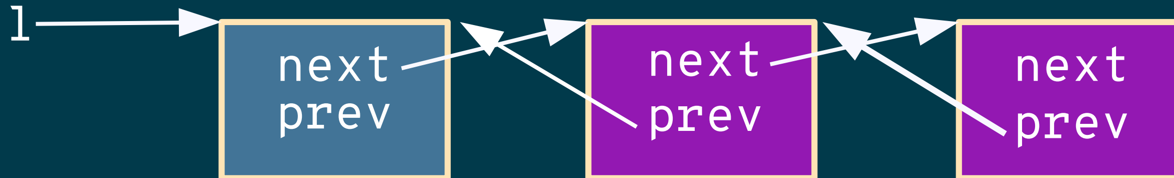
Different Allocators

```
alloc a;
auto l = new std::list<int,Alloc>(a);
l->push_back(1);
l->push_back(2);
```



Different Allocators

```
alloc a;
auto l = new std::list<int,Alloc>(a);
l->push_back(1);
l->push_back(2);
```



Memory Model

Concurrency in Standard

```
{
  int c = 0;
  auto f = [&](){ ++c; };
  std::thread t1(f), t2(f), t3(f);
  t1.join();
  t2.join();
  t3.join();
  std::cout << "c = " << c << std::endl;
}
```

Concurrency in Standard

```
{
  int c = 0;
  auto f = [&](){ ++c; };
  std::thread t1(f), t2(f), t3(f);
  t1.join();
  t2.join();
  t3.join();
  std::cout << "c = " << c << std::endl;
}
```

Undefined Behavior

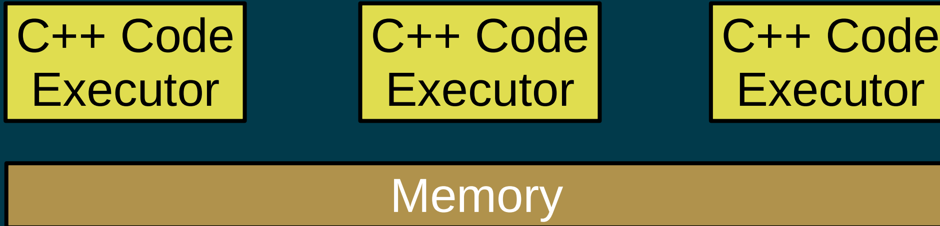
Abstract

C++ Code Executor

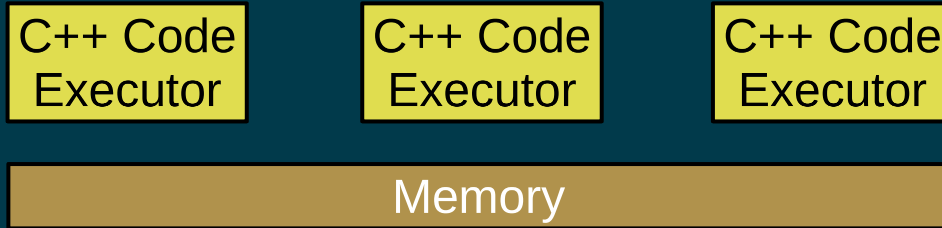
Memory

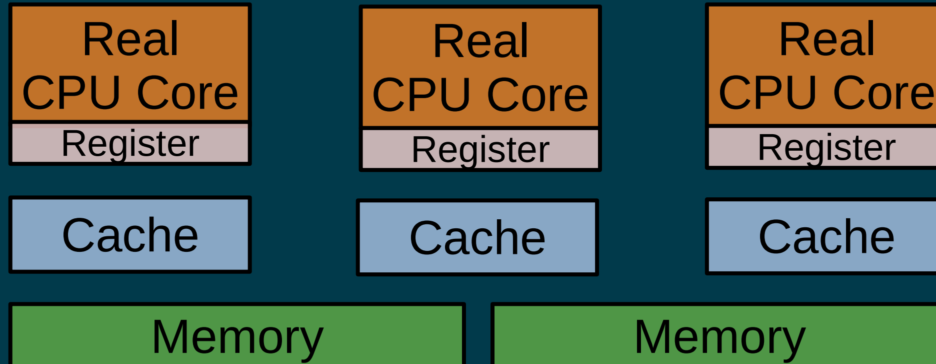
redhat.

Abstract

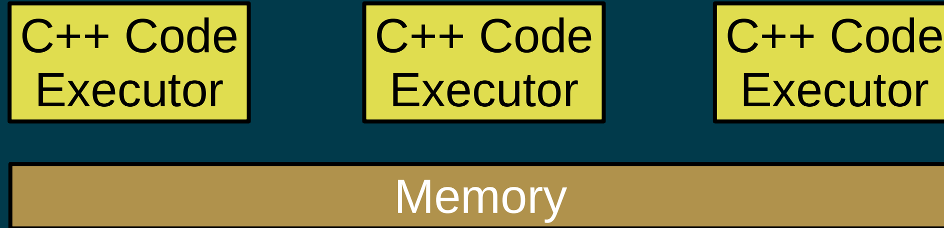


Abstract vs Real

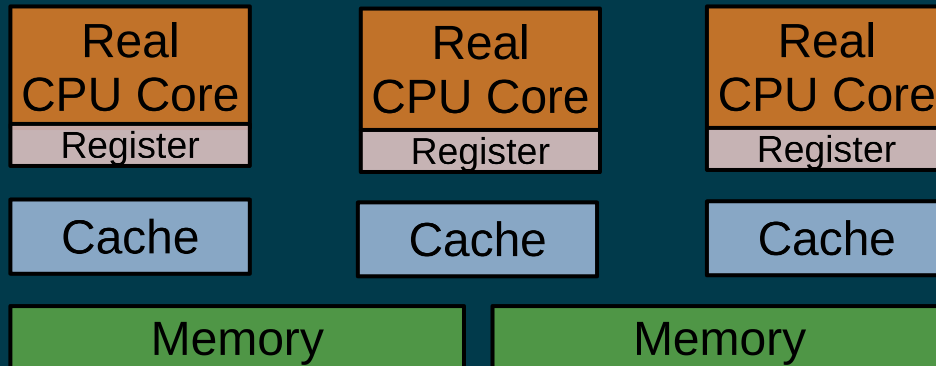




Abstract vs Real



Too expensive
to keep identical



Atomics

Used to be:

```
int cnt;

void count() {
  asm volatile("lock; addl $1,%m" : "=m" (cnt) : "m" (cnt));
}
```

Atomics

Now:

```
std::atomic<int> cnt;

void count() {
  ++cnt;
}
```

Atomics

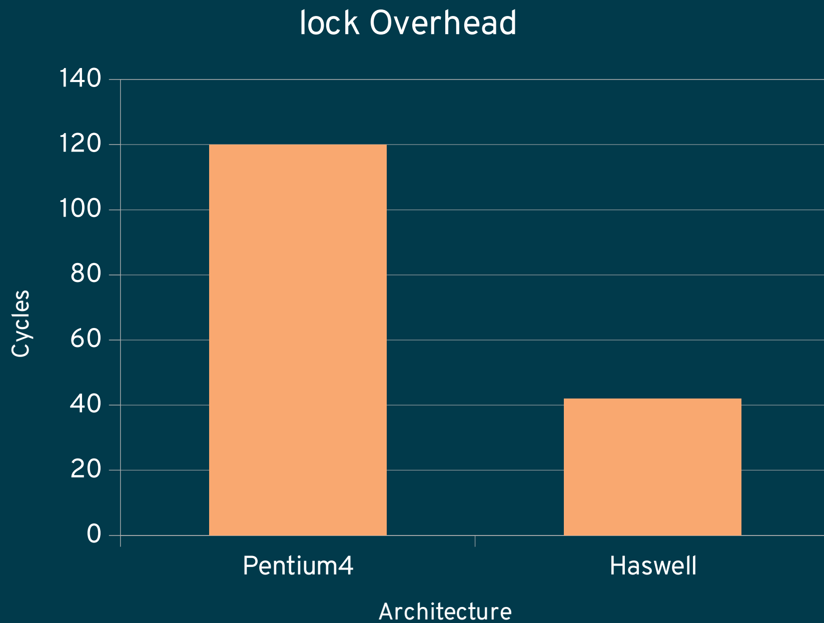
Now:

```
std::atomic<int> cnt;

void count() {
  ++cnt;
}
```

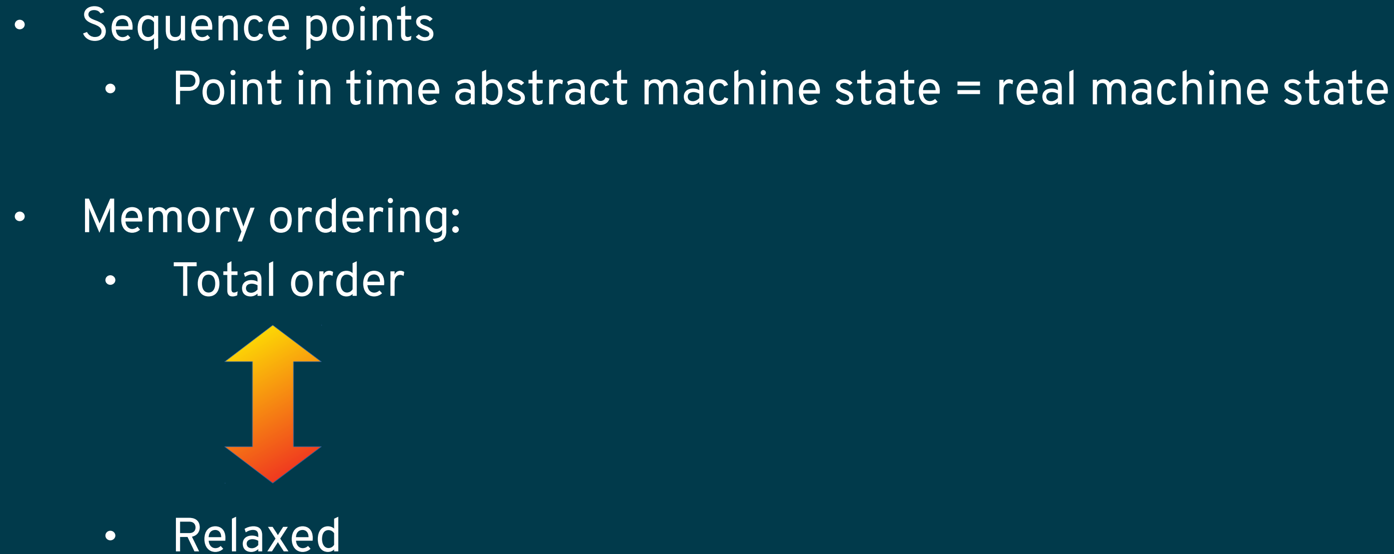
```
count():
   0: lock addl $0x1,cnt(%rip)
   8: retq
```

Atomics




```
count():
   0: lock addl $0x1,cnt(%rip)
   8: retq
```

Pick the Cost



THANK YOU

plus.google.com/+RedHat

facebook.com/redhatinc

linkedin.com/company/red-hat

twitter.com/RedHatNews

youtube.com/user/RedHatVideos