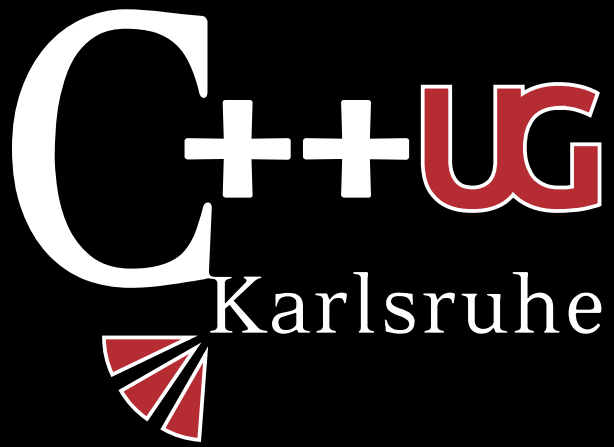
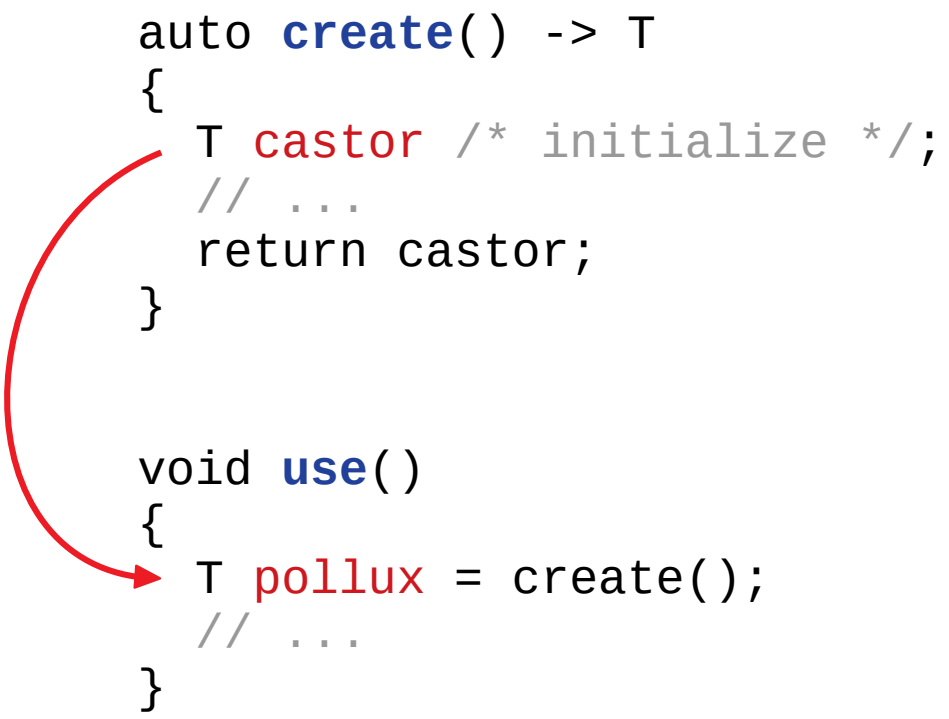
C++
Named Return Value Optimization

post. Robert .Schneider@hotmail.de



meetup.com/c-user-group-karlsruhe


auto create() -> T
{
T castor /* initialize */;
// ...
return castor;
}
void use()
{
T pollux = create();
// ...
}

variable  object  resource

```
{
  Type variable /* initialize */;
  // ...
  // ...
}
```

```
{
  Type variable /* initialize */;
  // ...
  return stuff;
}
```

```
{
  Type variable /* initialize */;
  // ...
  throw stuff;
}
```


variable
object
clean-up
scope left
destroyed
executed

```
class Type
{
  // ...

  ~Type()
  {
    // clean-up code
  }

};
```

```
{
  Type variable /* init */ ;
  Type other_var = variable;
  // ...
}
```


variable
object
clean-up
different
independent
independent

```
{
  Type variable /* init */ ;
  {
    Type other_var = variable;
    // ...
  }
  // ...
}
```

variable  object  resource

```
int
MyInt
array<int, 10>
vector<int>
```

```
{
  int castor = 0;
  int pollux = castor;

  assert(castor == pollux);

  castor = 1729;
  pollux = -1;

  assert(castor != pollux);
}
```

```
{
  vector<int> castor(1000);
  vector<int> pollux = castor;

  assert(castor == pollux);

  castor.at(42) = 1729;
  pollux.at(42) = -1;

  assert(castor != pollux);
}
```

```
int
MyInt
array<int, 10>
vector<int>
```

special

`vector<int>`

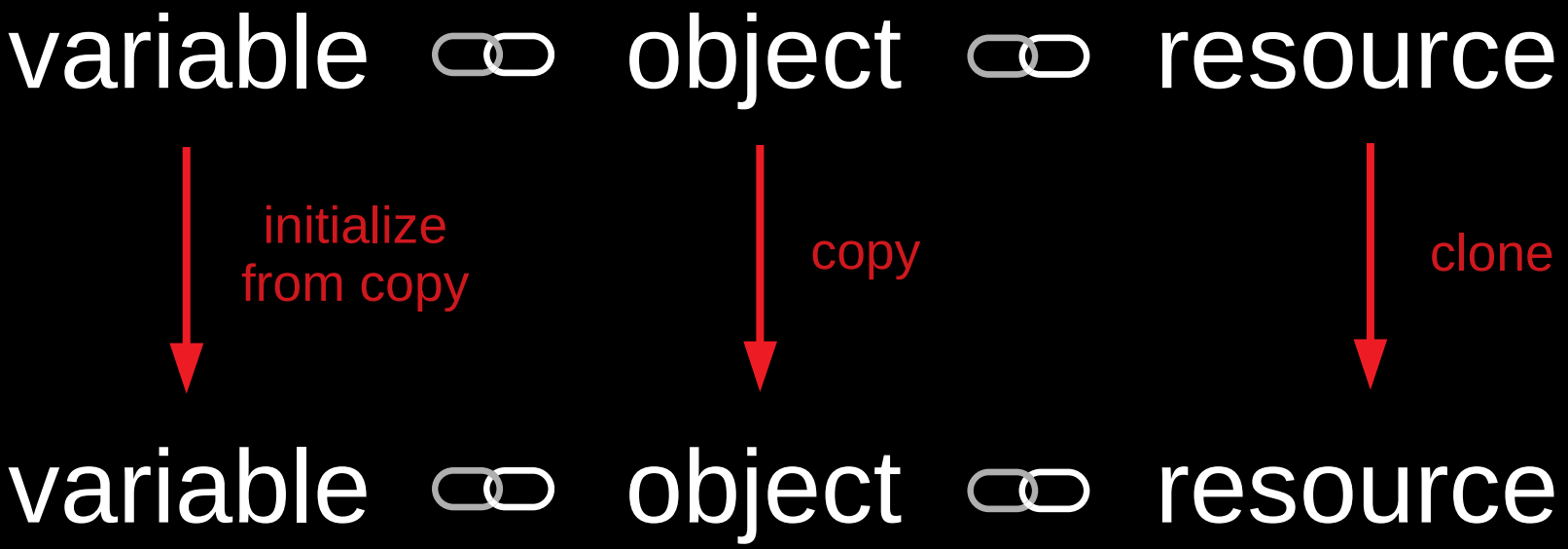
memory allocation

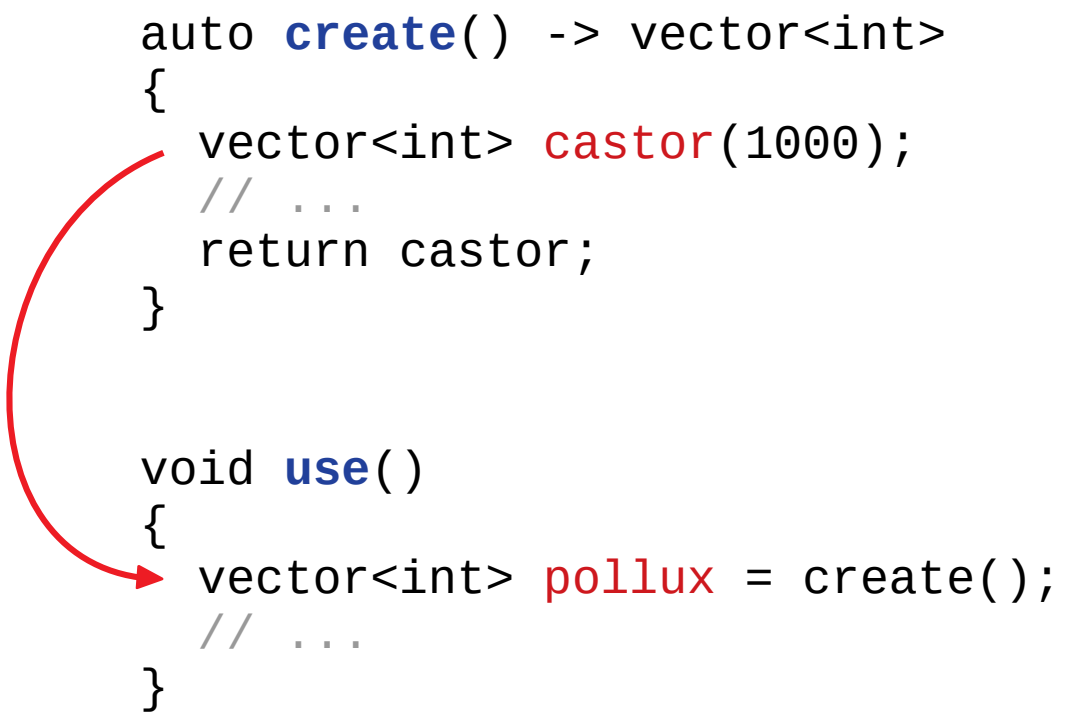
object

resource

```
{
  vector<int> castor(1000);
  vector<int> pollux = castor;

  assert(castor == pollux);

  castor.at(42) = 1729;
  pollux.at(42) = -1;

  assert(castor != pollux);
}
```


variable
object
resource
different
independent
independent

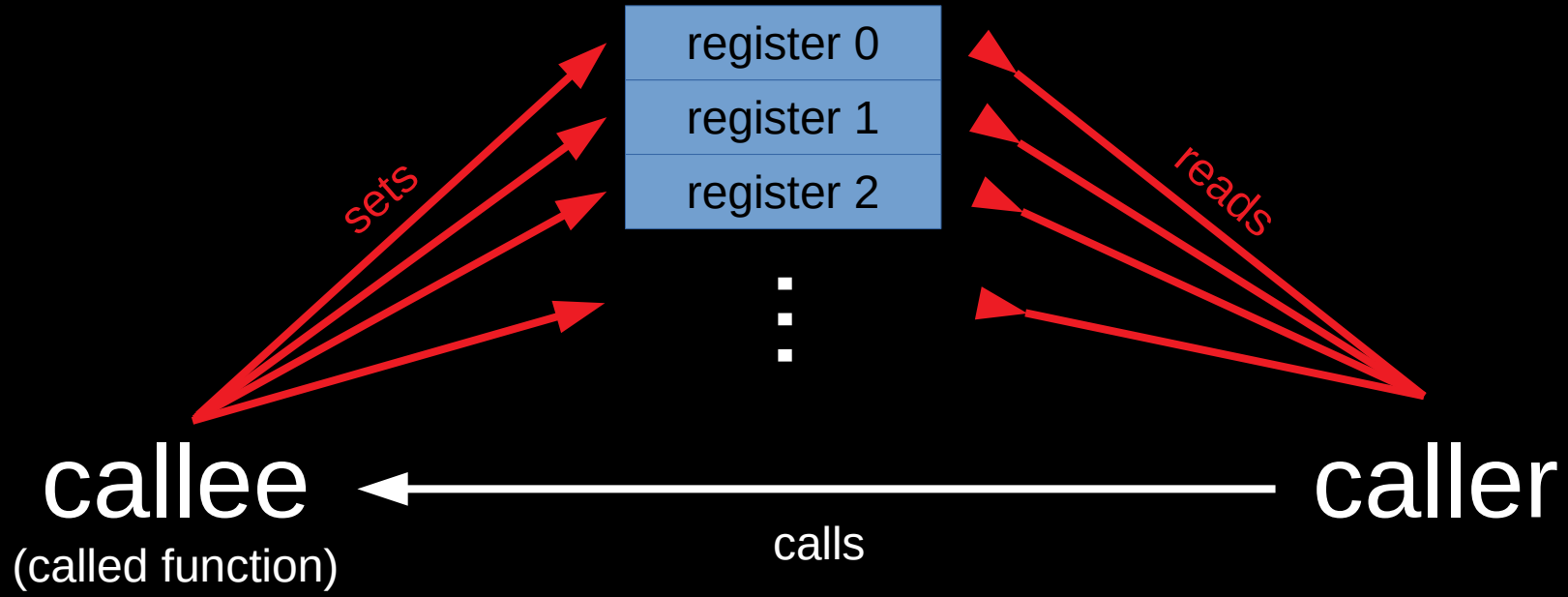

variable
object
resource
initialize
from copy
copy
clone
variable
object
resource


auto create() -> vector<int>
{
vector<int> castor(1000);
// ...
return castor;
}
void use()
{
vector<int> pollux = create();
// ...
}

variable  object  resource

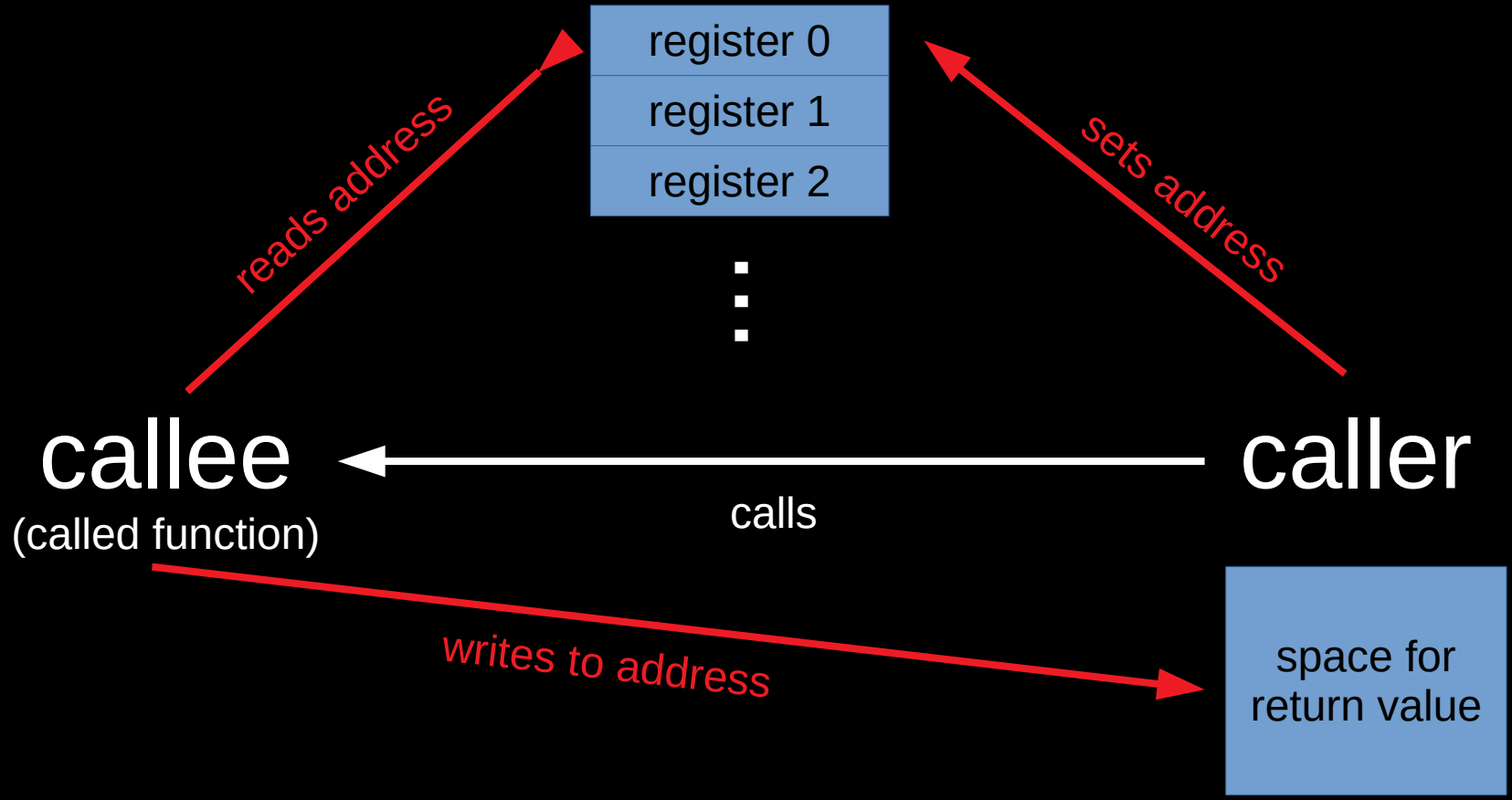
```
int_pair p() {
  int_pair r = {0};
  return r;
}
```

```
movl    $0, %eax
ret
```

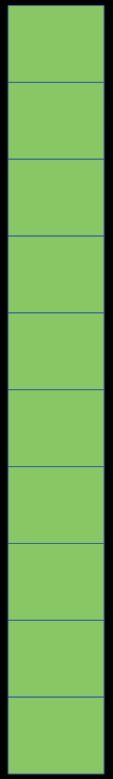

register 0
register 1
register 2
sets
reads
callee
(called function)
calls
caller

```
int_pair p() {
  int_pair r = {0};
  return r;
}

int_quadruple q() {
  int_quadruple r = {0};
  return r;
}
```

```
movl    $0, %eax
ret

movl    $0, %eax
movl    $0, %edx
ret
```


register 0
register 1
register 2
reads address
sets address
callee
(called function)
calls
caller
writes to address
space for
return value

RAM

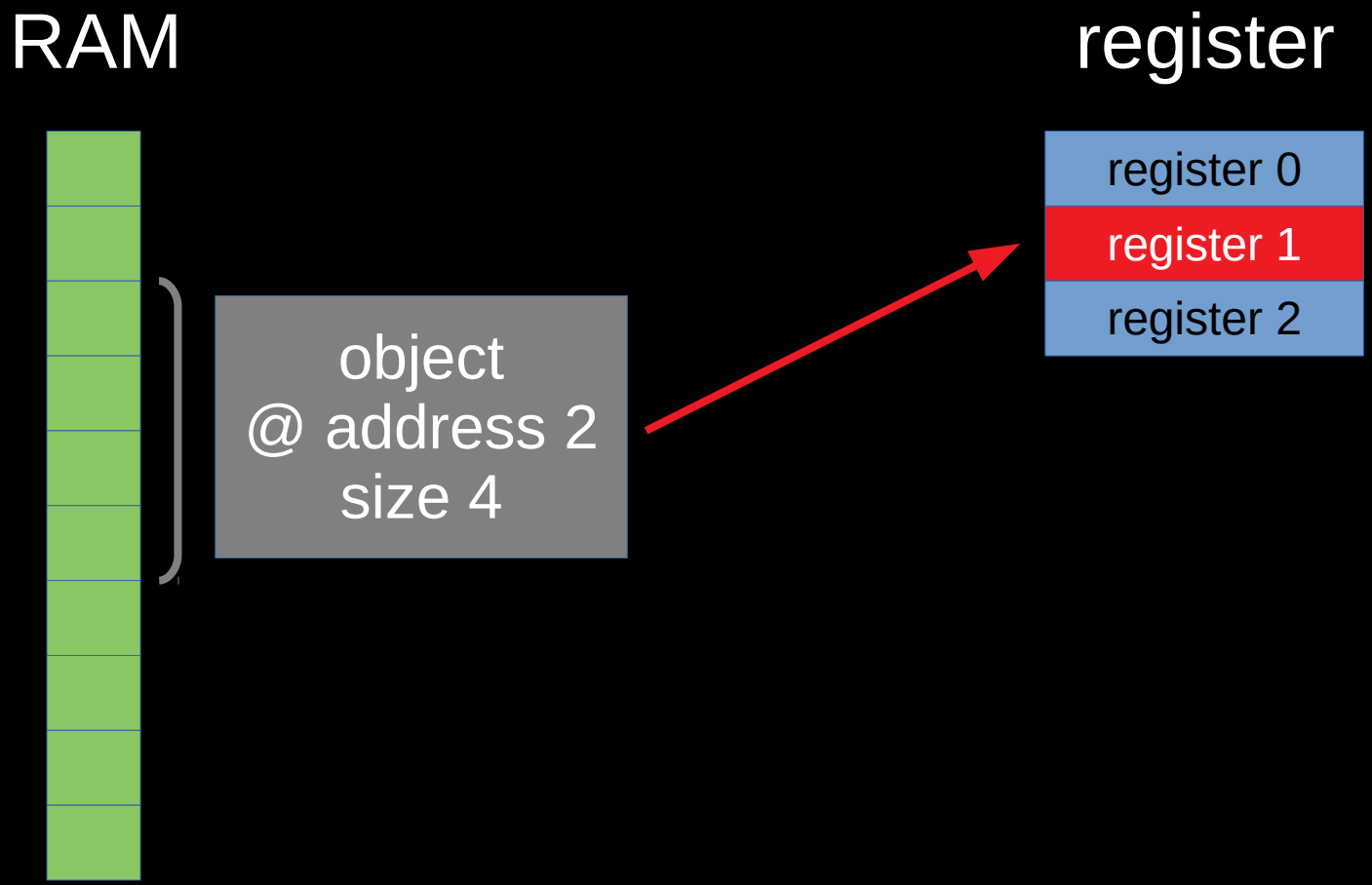


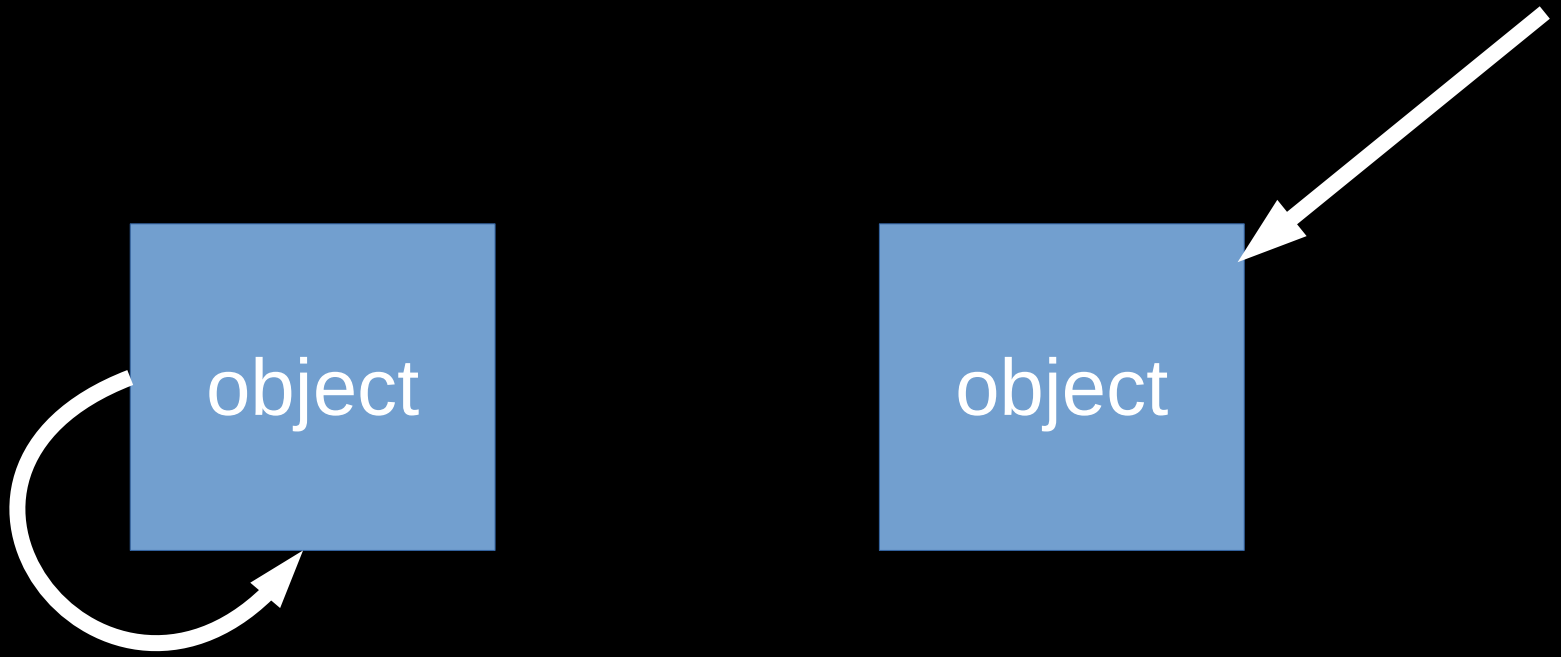
register

register 0
register 1
register 2

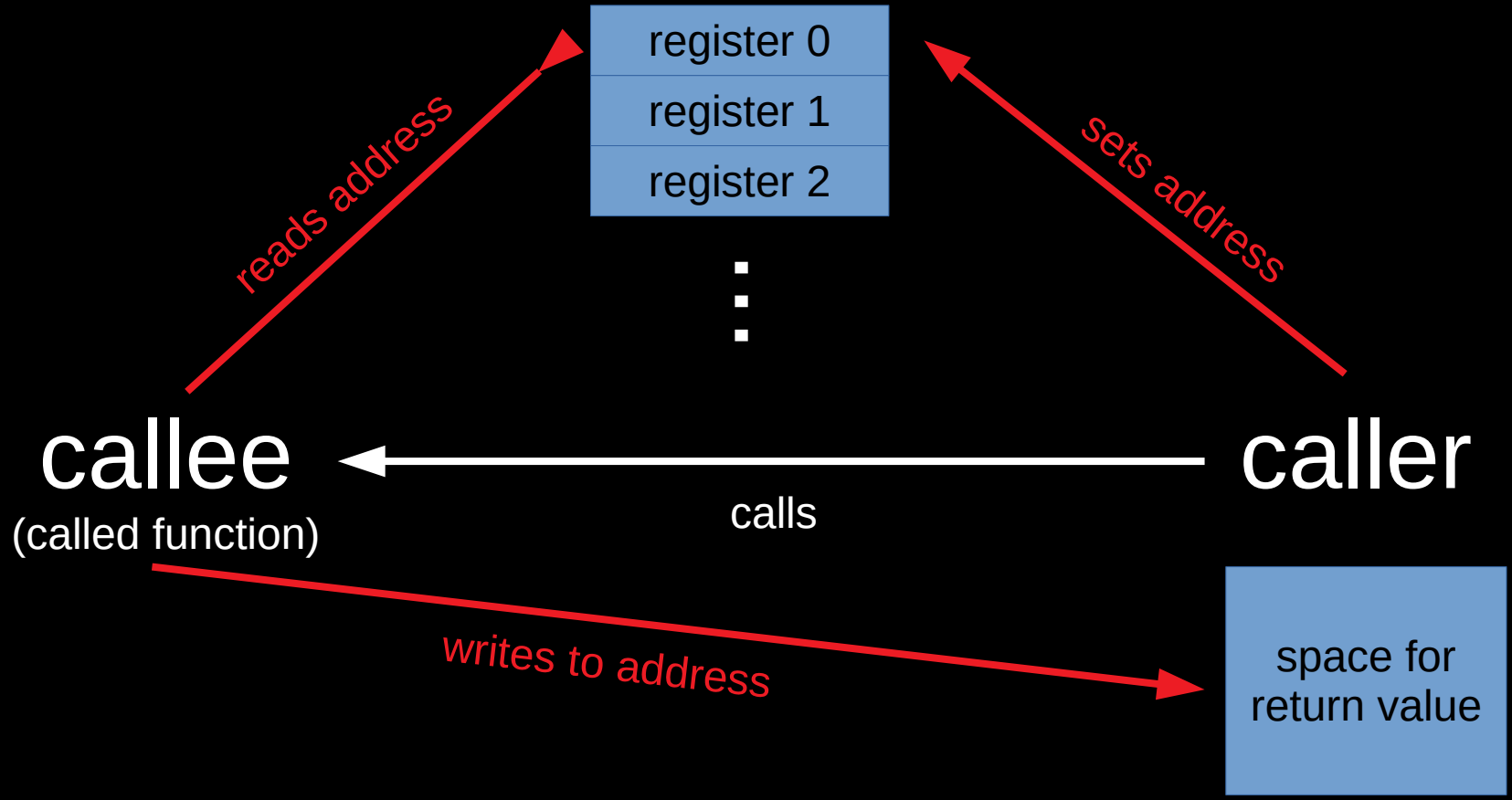
RAM

register


RAM
register
register 0
register 1
register 2
object
@ address 2
size 4


object
object

trivially copyable


register 0
register 1
register 2
reads address
sets address
callee
(called function)
calls
caller
writes to address
space for
return value

variable  object  resource

indirect return
for non-trivially-copyable

```
void use()
{

  vector<int> pollux = create();

  // ...
}

auto create() -> vector<int>
{
  vector<int> castor(1000);
  // ...
  return castor;
}
```

```
void use()
{
  buffer< vector<int> > future_pollux;
  create(&future_pollux);
  vector<int>& pollux = future_pollux;
  // ...
}

void create(buffer< vector<int> >* ret)
{
  vector<int> castor(1000);
  // ...
  new(ret) vector<int>(castor);
}
```

```
void create(
  buffer< vector<int> >* ret)
{                                   pushq   %rbx

  vector<int> castor(1000);         subq    $32, %rsp
                                    movq    %rdi, %rbx
                                    movl    $1000, %esi
                                    movq    %rsp, %rdi
                                    call    vector<int>::vector(int)

  new(ret) vector<int>(castor);     movq    %rsp, %rsi
                                    movq    %rbx, %rdi
                                    call    vector<int>
                                            ::vector(vector<int> const&)

}                                   movq    %rsp, %rdi
                                    call    vector<int>::~vector()
                                    movq    %rbx, %rax
                                    addq    $32, %rsp
                                    popq    %rbx
                                    ret
```

```
void create(
  buffer< vector<int> >* ret)
{                                   pushq   %rbx

  new(ret) vector<int>(1000);
                                    movq    %rdi, %rbx
                                    movl    $1000, %esi

                                    call    vector<int>::vector(int)









}                                   movq    %rbx, %rax
                                    popq    %rbx
                                    ret
```

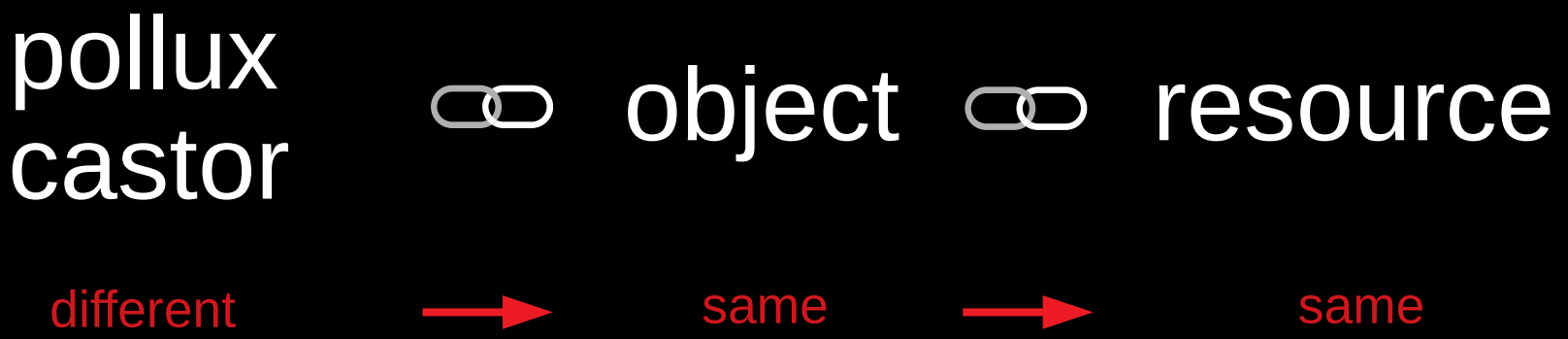
NRVO

=

indirect return

- unneccesary operations

```
void use()
{

  vector<int> pollux = create();

  // ...
}

auto create() -> vector<int>
{
  vector<int> castor(1000);
  // ...
  return castor;
}
```


pollux
castor
object
resource
different
same
same

observable behaviour is guaranteed

but not how it is achieved

```
int x = 0;

for(int i = 0; i < 100; ++i)
  x += i;

cout << x;
```

```
mov edi, offset std::cout
mov esi, 4950
call ostream::operator<<(int)
```

observable behaviour is guaranteed

but not how it is achieved

*except NRVO / copy elision

resource management
is
potentially observable

limits of NRVO

- local variable
- same type as return type

```
auto no_nrvo() -> vector<int>
{
  array< vector<int>, 2 > ogre;

  return ogre.at(0);
}
```

limits of NRVO

- local variable
- same type as return type
- complete object

```
auto nrvo(bool b) -> vector<int>
{
  vector<int> thor(500);
  vector<int> loki(900);

  if(b)
    return thor;
  else
    return loki;
}
```

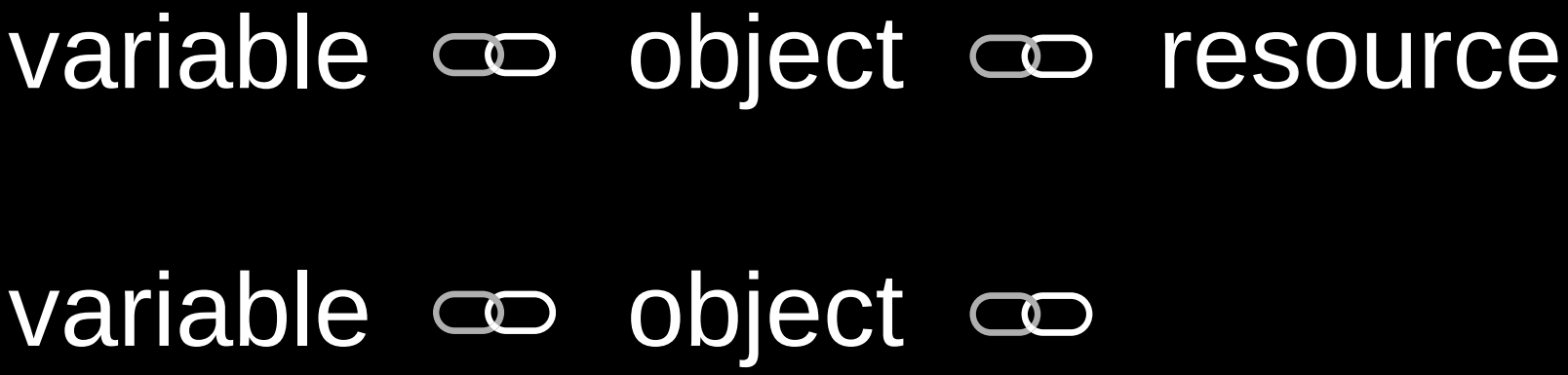
limits of NRVO

- local variable
- same type as return type
- complete object
- compiler smartness

```
auto nrvo() -> vector<int>
{
  vector<int> thor(500);
  vector<int> loki(900);

  if(read_char() == 'a')
    return thor;
  else
    return loki;
}
```

limits of NRVO

- local variable
- same type as return type
- complete object
- compiler smartness
- observable behaviour

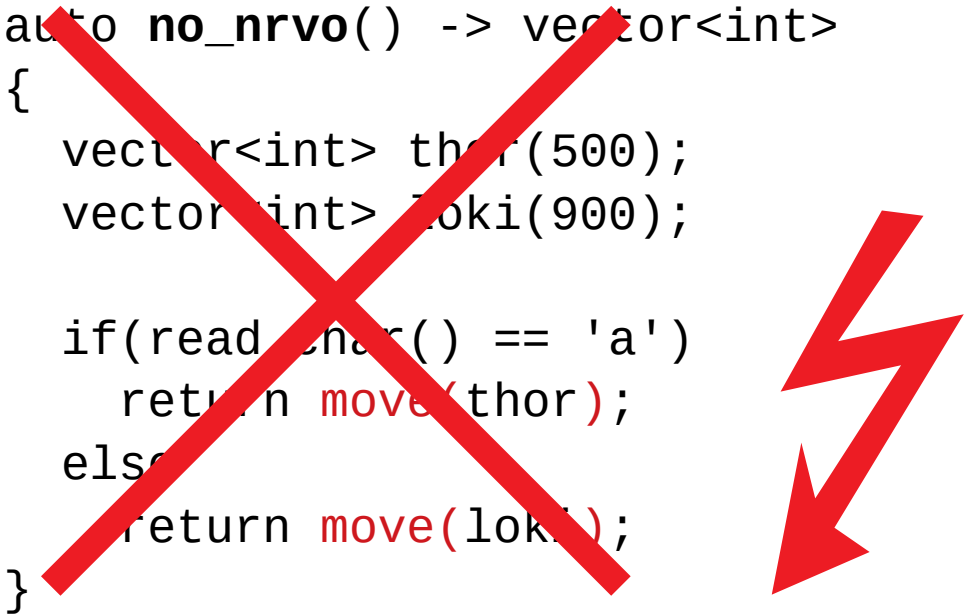

variable
object
resource
variable
object

moving is great! but...

- for some types it is still slow
- don’t want to write it explicitly

```
auto nrvo() -> vector<int>
{
  vector<int> thor(500);
  vector<int> loki(900);

  if(read_char() == 'a')
    return thor;
  else
    return loki;
}
```

NRVO > move > copy


auto no_nrvo() -> vector<int>
{
vector<int> thor(500);
vector<int> loki(900);
if(read_char() == 'a')
return move(thor);
else
return move(loki);
}