System Programming with C++

initializer_list<T>, smart pointers

std::vector<T>

You know `std::vector<T>` very well.

std::vector<T>

You know `std::vector<T>` very well. We've been implementing it during our lectures.

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        ...
    }
    ...
};
```

std::vector<T>

You know std::vector<T> very well. We've been implementing it during our lectures.

But there are still some (unclear) differences.

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        ...
    }
    ...
};
```

std::vector<T>

You know std::vector<T> very well. We've been implementing it during our lectures.

But there are still some (unclear) differences.

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        ...
    }
    ...
};
```

```
Vector<int> my_v{16};
// 16 capacity, 0 elements
```

std::vector<T>

You know std::vector<T> very well. We've been implementing it during our lectures.

But there are still some (unclear) differences.

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        ...
    }
    ...
};
```

```
Vector<int> my_v{16};
// 16 capacity, 0 elements

std::vector<int> std_v{16};
```

std::vector<T>

You know std::vector<T> very well. We've been implementing it during our lectures.

But there are still some (unclear) differences.

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        ...
    }
    ...
};
```

```
Vector<int> my_v{16};
// 16 capacity, 0 elements

std::vector<int> std_v{16};
// 1 element that equals 16
```

std::vector<T>

You know std::vector<T> very well. We've been implementing it during our lectures.

But there are still some (unclear) differences.

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        ...
    }
    ...
};
```

```
Vector<int> my_v{16};
// 16 capacity, 0 elements

std::vector<int> std_v{16};
// 1 element that equals 16
```

So, logic of constructor is different?

std::vector<T>

You know std::vector<T> very well. We’ve been implementing it during our lectures.

But there are still some (unclear) differences.

```cpp
std::vector<int> std_v{16, 12};
// ???
```

std::vector<T>

You know std::vector<T> very well. We've been implementing it during our lectures.

But there are still some (unclear) differences.

```
std::vector<int> std_v{16, 12};
// 16 12
```

std::vector<T>

You know std::vector<T> very well. We've been implementing it during our lectures.

But there are still some (unclear) differences.

```cpp
std::vector<int> std_v{16, 12};
// 16 12

std::vector<int> std_v2(16, 12);
// ???
```

std::vector<T>

You know std::vector<T> very well. We've been implementing it during our lectures.

But there are still some (unclear) differences.

```
std::vector<int> std_v{16, 12};
// 16 12

std::vector<int> std_v2(16, 12);
// 12 12 12 12 12 12 12 12 12 12 12 12 ... why?
```

Initialization in C++ (first approximation)

1. Default initialization in C++ ✓
   a. same UB for primitives
   b. default ctors for classes

2. Value initialization ✓ ✗ — ok for primitives, nightmare for classes, avoid if possible

3. Direct initialization ✓ — use (args) or {args}

4. Copy initialization ✓ — implicit call of ctrs here, use explicit to avoid

5. Aggregate initialization ✓ — works for aggregates, use copy initialization for all fields

Initialization in C++ (first approximation)

1. Default initialization in C++
   a. same UB for primitives
   b. default ctors for classes
2. Value initialization

   ok for primitives,
   nightmare for classes,
   avoid if possible
3. Direct initialization

   use (args) or {args}
4. Copy initialization

   implicit call of ctrs here,
   use explicit to avoid
5. Aggregate initialization

   works for aggregates, use
   copy initialization for all
   fields
6. List initialization (direct and copy)

std::initializer_list<T>

```
auto int_il = {1, 2, 3};
```

std::initializer_list<T>

```
auto int_il = {1, 2, 3};
// std::initializer_list<int>
```

special class from std
for such scenarios

std::initializer_list<T>

```
auto int_il = {1, 2, 3};
// std::initializer_list<int>
```

special class from std
for such scenarios

```
Vector(std::initializer_list<T> il): Vector(16) {
    for(auto&& e: il) {
        push(e);
    }
}
```

std::initializer_list<T>

```
auto int_il = {1, 2, 3};
// std::initializer_list<int>
```

special class from std for such scenarios

```
Vector(std::initializer_list<T> il): Vector(16) {
    for(auto&& e: il) {
        push(e);
    }
}
```

inside you can use it as a usual sequence container

std::initializer_list<T>

```
auto int_il = {1, 2, 3};
// std::initializer_list<int>
```

special class from std
for such scenarios

```
Vector(std::initializer_list<T> il): Vector(16) {
    for(auto&& e: il) {
        push(e);
    }
}
```

std::initializer_list<T>

```
auto int_il = {1, 2, 3};
// std::initializer_list<int>
```

special class from std
for such scenarios

```
Vector(std::initializer_list<T> il): Vector(16) {
   for(auto&& e: il) {
       push(e);
   }
}

Vector<int> vi(int_il);
```

std::initializer_list<T>

```cpp
auto int_il = {1, 2, 3};
// std::initializer_list<int>
```

special class from std
for such scenarios

```cpp
Vector(std::initializer_list<T> il): Vector(16) {
   for(auto&& e: il) {
       push(e);
   }
}
```

```cpp
Vector<int> vi(int_il);
```

direct initialization

std::initializer_list<T>

```
auto int_il = {1, 2, 3};
// std::initializer_list<int>
```

special class from std for such scenarios

```
Vector(std::initializer_list<T> il): Vector(16) {
    for(auto&& e: il) {
        push(e);
    }
}
```

```
Vector<int> vi(int_il);
```

direct initialization

```
Vector<int> vi2 = int_il;
```

copy initialization

std::initializer_list<T>

```
auto int_il = {1, 2, 3};
// std::initializer_list<int>
```

special class from std for such scenarios

```
Vector(std::initializer_list<T> il): Vector(16) {
   for(auto&& e: il) {
       push(e);
   }
}
```

```
Vector<int> vi{1, 2, 3};
```

direct initialization

```
Vector<int> vi2 = {1, 2, 3};
```

copy initialization

std::initializer_list<T>

```
auto int_il = {1, 2, 3};
// std::initializer_list<int>
```

special class from std for such scenarios

```
Vector(std::initializer_list<T> il): Vector(16) {
    for(auto&& e: il) {
        push(e);
    }
}
```

```
Vector<int> vi{1, 2, 3};
```

direct list initialization

```
Vector<int> vi2 = {1, 2, 3};
```

copy list initialization

std::initializer_list<T>

```
template <typename T>
class Vector {
   Vector(size_t initial_capacity): cap_(initial_capacity) { }

   Vector(std::initializer_list<T> il): Vector(16) {
       for(auto&& e: il) {
           push(e);
       }
   }
};
```

std::initializer_list<T>

```
template <typename T>
class Vector {
   Vector(size_t initial_capacity): cap_(initial_capacity) { }

   Vector(std::initializer_list<T> il): Vector(16) {
       for(auto&& e: il) {
           push(e);
       }
   }
};

Vector<int> vi{4};
```

std::initializer_list<T>

```
template <typename T>
class Vector {
   Vector(size_t initial_capacity): cap_(initial_capacity) { }

   Vector(std::initializer_list<T> il): Vector(16) {
       for(auto&& e: il) {
           push(e);
       }
   }
};

Vector<int> vi{4};     ←——     which constructor to choose?
```

std::initializer_list<T>

```
template <typename T>
class Vector {
   Vector(size_t initial_capacity): cap_(initial_capacity) { }

   Vector(std::initializer_list<T> il): Vector(16) {
       for(auto&& e: il) {
           push(e);
       }
   }
};

Vector<int> vi{4};
```

which constructor to choose?

```
// Vector with cap_ = 16 and one element which one is 4 is created
```

std::vector<T>

You know std::vector<T> very well. We've been implementing it during our lectures.

But there are still some (unclear) differences.

```
std::vector<int> std_v{16, 12};
// 16 12

std::vector<int> std_v2(16, 12);
// 12 12 12 12 12 12 12 12 12 12 12 12 ... why?
```

std::vector<T>

You know std::vector<T> very well. We’ve been implementing it during our lectures.

But there are still some (unclear) differences.

```
std::vector<int> std_v{16, 12};
// 16 12

std::vector<int> std_v2(16, 12);
// 12 12 12 12 12 12 12 12 12 12 12 12 ... because of uniform initialization
```

~~Unicorn~~ Uniform initialization

```
Vector<int> vi{4};
```

Which type of initialization we see?

~~Unicorn~~ Uniform initialization

```
Vector<int> vi{4};
```

Which type of initialization we see?

1. If type Vector is aggregate => aggregate initialization

~~Unicorn~~ Uniform initialization

```
Vector<int> vi{4};
```

Which type of initialization we see?

1. If type Vector is aggregate => aggregate initialization
2. If there there is constructor with initializer_list argument => list initialization (this constructor will be taken)

~~Unicorn~~ Uniform initialization

```
Vector<int> vi{4};
```

Which type of initialization we see?

1. If type Vector is aggregate => aggregate initialization

2. If there there is constructor with initializer_list argument => list initialization (this constructor will be taken)

3. Otherwise, constructor with 1 int argument will be taken.

Initialization in C++ (first approximation)

1. Default initialization in C++ ✓
   a. same UB for primitives
   b. default ctors for classes
2. Value initialization ✓ ✗ — ok for primitives, nightmare for classes, avoid if possible
3. Direct initialization ✓ — use (args) or {args}
4. Copy initialization ✓ — implicit call of ctrs here, use explicit to avoid
5. Aggregate initialization ✓ — works for aggregates, use copy initialization for all fields
6. List initialization (direct and copy)

Initialization in C++ (second approximation)

1. Default initialization in C++
   a. same UB for primitives
   b. default ctors for classes
2. Value initialization — ok for primitives, nightmare for classes, avoid if possible
3. Direct initialization — use (args) or {args}
4. Copy initialization — implicit call of ctrs here, use explicit to avoid
5. Aggregate initialization — works for aggregates, use copy initialization for all fields
6. List initialization — uniform initialization

~~Unicorn~~ Uniform initialization

```
Vector<int> vi{4};
```

Which type of initialization we see?

1. If type Vector is aggregate => aggregate initialization

2. If there there is constructor with initializer_list argument => list initialization (this constructor will be taken)

3. Otherwise, constructor with 1 int argument will be taken.

~~Unicorn~~ Uniform initialization

So, when we are talking about classes with initializer_list constructors, the good idea is to use **(...)** not **{...}** for other constructors.

```
Vector<int> vi{4};
```

Which type of initialization we see?

1. If type Vector is aggregate => aggregate initialization
2. If there there is constructor with initializer_list argument => list initialization (this constructor will be taken)
3. Otherwise, constructor with 1 int argument will be taken.

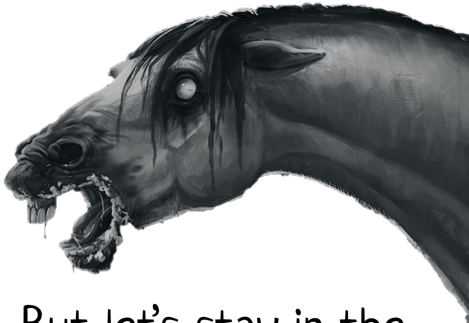
~~Unicorn~~ Uniform initialization

```
Vector<int> vi{4};
```

Which type of initialization we see?

1. If type Vector is aggregate => aggregate initialization
2. Special case for empty brackets => value init
3. If there there is constructor with initializer_list argument => list initialization (this constructor will be taken)
4. Otherwise, constructor with 1 int argument will be taken.

~~Unicorn~~ Uniform initialization



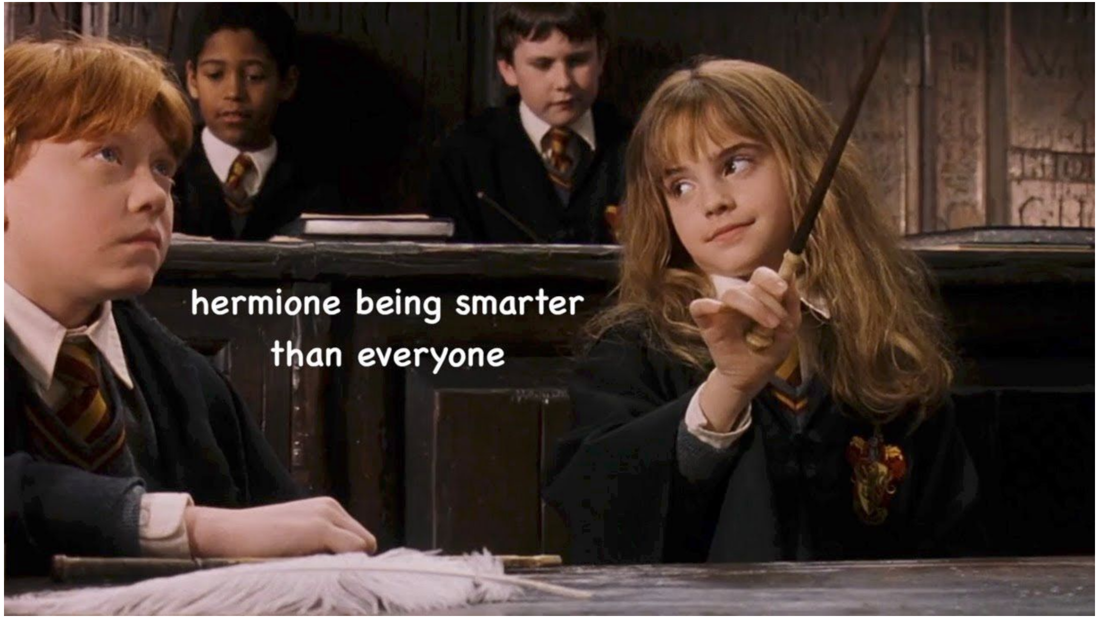
But let's stay in the world of unicorns.

```
Vector<int> vi{4};
```

Which type of initialization we see?

1. If type Vector is aggregate => aggregate initialization
2. Special case for empty brackets => value init
3. If there there is constructor with initializer_list argument => list initialization (this constructor will be taken)
4. Otherwise, constructor with 1 int argument will be taken.

Smart Pointers



Smart Pointers

C++ is language with manual memory management.

Smart Pointers

C++ is language with manual memory management.

Which problems does it bring?

Smart Pointers

C++ is language with manual memory management.

Which problems does it bring?

- Dangling pointers/references

Smart Pointers

C++ is language with manual memory management.

Which problems does it bring?

- Dangling pointers/references

- Memory leaks

RAII

What's wrong with this code?

```
struct Triple { int x; int y; int z; };

int baz() {
    Triple* t = new Triple{13, 42, 1};
    int val =  bar();
    if (val > 13) {
        delete t;
        return 0;
    }
    // ... some code that uses t ...
    int result = val + t->x + t->z;
    delete t;
    return result;
}
```

What if bar throws an exception? Memory leak!

copy-paste

copy-paste

```cpp
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }
   ScopedPointer(const ScopedPointer& other) = delete;
   ScopedPointer(ScopedPointer&& other) { … }
   ~ScopedPointer() { delete pointer; }
   ...
};

int foo() {
    ScopedPointer sp{new Triple{13, 42, 1}};
    if (...) {
        throw "ooops";
    }
    return sp->x + sp->z;
}
```

One of the variants
from your NSTT #5.

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }
   ScopedPointer(const ScopedPointer& other) = delete;
   ScopedPointer(ScopedPointer&& other) { … }
   ~ScopedPointer() { delete pointer; }
   ...
};

int foo() {
    ScopedPointer sp{new Triple{13, 42, 1}};
    if (...) {
        throw "ooops";
    }
    return sp->x + sp->z;
}
```

destructor is called,
memory is freed => no leak

std::unique_ptr

```
#include <memory>

int foo() {
    std::unique_ptr<Triple> ptr{new Triple{1, 2, 3}};
    if (ptr->y == 42) {
        throw "ooops";
    }
    return ptr->x + ptr->z;
}
```

std::unique_ptr

```cpp
#include <memory>

int foo() {
   std::unique_ptr<Triple> ptr{new Triple{1, 2, 3}};
   if (ptr->y == 42) {
       throw "ooops";
   }
   return ptr->x + ptr->z;
}
```

destructor is called => memory is freed

std::unique_ptr

```
#include <memory>

int foo() {
   Triple* raw_triple = new Triple{1, 2, 3};
   std::unique_ptr<Triple> ptr{raw_triple};
   if (ptr->y == 42) {
       throw "ooops";
   }
   return ptr->x + ptr->z;
}
```

std::unique_ptr

```
#include <memory>

int foo() {
   Triple* raw_triple = new Triple{1, 2, 3};
   std::unique_ptr<Triple> ptr1{raw_triple};
   {
       std::unique_ptr<Triple> ptr2{raw_triple};
   }
   if (ptr1->y == 42) {
       throw "ooops";
   }
   return ptr1->x + ptr1->z;
}
```

std::unique_ptr

```cpp
#include <memory>

int foo() {
    Triple* raw_triple = new Triple{1, 2, 3};
    std::unique_ptr<Triple> ptr1{raw_triple};
    {
        std::unique_ptr<Triple> ptr2{raw_triple};
    }
    if (ptr1->y == 42) {
        throw "ooops";
    }
    return ptr1->x + ptr1->z;
}
```

UB

double free

Takeaways:

std::unique_ptr

1. Unique pointer is for unique ownership!

```
#include <memory>


int foo() {
    Triple* raw_triple = new Triple{1, 2, 3};
    std::unique_ptr<Triple> ptr1{raw_triple};
    {
        std::unique_ptr<Triple> ptr2{raw_triple};
    }
    if (ptr1->y == 42) {
        throw "ooops";
    }
    return ptr1->x + ptr1->z;
}
```

UB

double free

std::unique_ptr

Takeaways:

1. Unique pointer is for unique ownership!
2. If we keep working with raw pointers it is dangerous.

```
#include <memory>

int foo() {
    Triple* raw_triple = new Triple{1, 2, 3};
    std::unique_ptr<Triple> ptr1{raw_triple};
    {
        std::unique_ptr<Triple> ptr2{raw_triple};
    }
    if (ptr1->y == 42) {
        throw "ooops";
    }
    return ptr1->x + ptr1->z;
}
```

UB

double free

std::unique_ptr

```cpp
#include <memory>

int foo() {
   Triple* raw_triple = new Triple{1, 2, 3};
   std::unique_ptr<Triple> ptr1{raw_triple};
   if (ptr1->y == 42) {
       throw "ooops";
   }
   return ptr1->x + ptr1->z;
}
```

std::unique_ptr

```
#include <memory>

int foo() {
    auto ptr = std::make_unique<Triple>(1, 2, 3);
    if (ptr->y == 42) {
        throw "ooops";
    }
    return ptr->x + ptr->z;
}
```

std::unique_ptr

No more clues about raw pointers! Working only with smart one.

```cpp
#include <memory>


int foo() {
    auto ptr = std::make_unique<Triple>(1, 2, 3);
    if (ptr->y == 42) {
        throw "ooops";
    }
    return ptr->x + ptr->z;
}
```

std::unique_ptr

```
#include <memory>

int foo() {
   auto ptr = std::make_unique<Triple>(1, 2, 3);
   if (ptr->y == 42) {
       throw "ooops";
   }
   return ptr->x + ptr->z;
}
```

No more clues about raw pointers! Working only with smart one.

Perfect forwarding of arguments, so zero cost abstraction.

std::unique_ptr

```
#include <memory>

int foo() {
   auto ptr = std::make_unique<Triple>(1, 2, 3);
   if (ptr->y == 42) {
       throw "ooops";
   }
   return ptr->x + ptr->z;
}
```

No more clues about raw pointers! Working only with smart one.

Perfect forwarding of arguments, so zero cost abstraction.

But what if I want to pass it as a param to other function?

std::unique_ptr

```cpp
#include <memory>

template <typename T>
void print(std::unique_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}

int foo() {
   auto ptr = std::make_unique<Triple>(1, 2, 3);
   if (ptr->y == 42) {
       throw "ooops";
   }
   return ptr->x + ptr->z;
}
```

std::unique_ptr

```
#include <memory>

template <typename T>
void print(std::unique_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}

int foo() {
    auto ptr = std::make_unique<Triple>(1, 2, 3);
    print(ptr);
    if (ptr->y == 42) {
        throw "ooops";
    }
    return ptr->x + ptr->z;
}
```

std::unique_ptr

```
#include <memory>

template <typename T>
void print(std::unique_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}

int foo() {
   auto ptr = std::make_unique<Triple>(1, 2, 3);
   print(ptr);
   if (ptr->y == 42) {
       throw "ooops";
   }
   return ptr->x + ptr->z;
}
```

Compilation error, no copy constructor!

std::unique_ptr

```
#include <memory>

template <typename T>
void print(std::unique_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}

int foo() {
   auto ptr = std::make_unique<Triple>(1, 2, 3);
   print(std::move(ptr));
   if (ptr->y == 42) {
       throw "ooops";
   }
   return ptr->x + ptr->z;
}
```

std::unique_ptr

```cpp
#include <memory>

template <typename T>
void print(std::unique_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}

int foo() {
   auto ptr = std::make_unique<Triple>(1, 2, 3);
   print(std::move(ptr));
   if (ptr->y == 42) {
       throw "ooops";
   }
   return ptr->x + ptr->z;
}
```

UB

double free!

std::unique_ptr

```
#include <memory>

template <typename T>
void print(std::unique_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}

int foo() {
   auto ptr = std::make_unique<Triple>(1, 2, 3);
   print(std::move(ptr));
   if (ptr->y == 42) {
       throw "ooops";
   }
   return ptr->x + ptr->z;
}
```

During print the argument had ownership over memory, but it died at the end of the scope.

UB

double free!

What do you think about this?

std::unique_ptr

```
#include <memory>

template <typename T>
void print(std::unique_ptr<T>& ptr) {
   std::cout << *ptr << std::endl;
}

int foo() {
   auto ptr = std::make_unique<Triple>(1, 2, 3);
   print(ptr);
   if (ptr->y == 42) {
       throw "ooops";
   }
   return ptr->x + ptr->z;
}
```

std::unique_ptr

What do you think about this?

This will work, but having a pointer or reference to unique_pointer is straight way to hell and dangling pointers or refs. Avoid it.

```
#include <memory>

template <typename T>
void print(std::unique_ptr<T>& ptr) {
    std::cout << *ptr << std::endl;
}

int foo() {
    auto ptr = std::make_unique<Triple>(1, 2, 3);
    print(ptr);
    if (ptr->y == 42) {
        throw "ooops";
    }
    return ptr->x + ptr->z;
}
```

std::unique_ptr

```cpp
#include <memory>

template <typename T>
void print(std::unique_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}

int foo() {
    auto ptr = std::make_unique<Triple>(1, 2, 3);
    print(std::move(ptr));
    if (ptr->y == 42) {
        throw "ooops";
    }
    return ptr->x + ptr->z;
}
```

std::unique_ptr

```cpp
#include <memory>

template <typename T>
auto print(std::unique_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
    return std::move(ptr);
}

int foo() {
    auto ptr = std::make_unique<Triple>(1, 2, 3);
    ptr = print(std::move(ptr));
    if (ptr->y == 42) {
        throw "ooops";
    }
    return ptr->x + ptr->z;
}
```

std::unique_ptr

```
#include <memory>

template <typename T>
auto print(std::unique_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
    return std::move(ptr);
}

int foo() {
    auto ptr = std::make_unique<Triple>(1, 2, 3);
    ptr = print(std::move(ptr));
    if (ptr->y == 42) {
        throw "ooops";
    }
    return ptr->x + ptr->z;
}
```

Better: we will transfer ownership back, so, this is correct.

std::unique_ptr

```
#include <memory>

template <typename T>
auto print(std::unique_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
    return std::move(ptr);
}

int foo() {
    auto ptr = std::make_unique<Triple>(1, 2, 3);
    ptr = print(std::move(ptr));
    if (ptr->y == 42) {
        throw "ooops";
    }
    return ptr->x + ptr->z;
}
```

Better: we will transfer ownership back, so, this is correct.

But too verbose and inconvenient. We will find a better way.

deleters

One of the variants
from your NSTT #5.

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }
   ScopedPointer(const ScopedPointer& other) = delete;
   ScopedPointer(ScopedPointer&& other) { … }
   ~ScopedPointer() { delete pointer; }
   ...
};

void foo() {
    ScopedPointer sp{new Triple{13, 42, 1}};
    if (...) {
        throw "ooops";
    }
    return sp->x + sp->z;
}
```

destructor is called,
memory is freed => no leak

One of the variants
from your NSTT #5.

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }
   ScopedPointer(const ScopedPointer& other) = delete;
   ScopedPointer(ScopedPointer&& other) { … }
   ~ScopedPointer() { delete pointer; }
   ...
};
```

Are you sure that this is correct?

```
void foo() {
    ScopedPointer sp{new Triple{13, 42, 1}};
    if (...) {
        throw "ooops";
    }
    return sp->x + sp->z;
}
```

destructor is called,
memory is freed => no leak

One of the variants
from your NSTT #5.

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }
   ScopedPointer(const ScopedPointer& other) = delete;
   ScopedPointer(ScopedPointer&& other) { … }
   ~ScopedPointer() { delete pointer; }
   ...
};
```

Are you sure that this is correct?
What about arrays?

```
void foo() {
    ScopedPointer sp{new Triple{13, 42, 1}};
    if (...) {
        throw "ooops";
    }
    return sp->x + sp->z;
}
```

destructor is called,
memory is freed => no leak

deleters

```
int foo() {
   auto ptr = std::make_unique<Triple>(1, 2, 3);
   if (ptr->y == 42) {
       throw "ooops";
   }
   return ptr->x + ptr->z;
}
```

deleters

```
int foo() {
   auto triples = new Triple[10];
   auto ptr = std::unique_ptr<Triple[]>(triples);
   return 42;
}
```

deleters

```cpp
int foo() {
    auto triples = new Triple[10];
    auto ptr = std::unique_ptr<Triple[]>(triples);
    return 42;
}
```

What will happen?

deleters

```
int foo() {
    auto triples = new Triple[10];
    auto ptr = std::unique_ptr<Triple[]>(triples);
    return 42;
}
```

What will happen?

`delete[] triples => 10 destructors of elements`

deleters

```
int foo() {
   auto triples = new Triple[10];
   auto ptr = std::unique_ptr<Triple>(triples);
   return 42;
}
```

What will happen?

deleters

```cpp
int foo() {
    auto triples = new Triple[10];
    auto ptr = std::unique_ptr<Triple>(triples);
    return 42;
}
```

What will happen?

deleters

```
int foo() {
    auto triples = new Triple[10];
    auto ptr = std::unique_ptr<Triple>(triples);
    return 42;
}
```

What will happen?

delete triples => UB (most probably segfault)

deleters

```
int foo() {
   auto triples = new Triple[10];
   auto ptr = std::unique_ptr<Triple[]>(triples);
   return 42;
}
```

Looks like it works differently for arrays and usual pointers.

How to implement?

deleters

```
int foo() {
    auto triples = new Triple[10];
    auto ptr = std::unique_ptr<Triple[]>(triples);
    return 42;
}
```

Looks like it works differently for arrays and usual pointers.

How to implement? Template specialization!

One of the variants from your NSTT #5.

```
template<typename T>
class ScopedPointer {
    T* pointer;

public:
    ScopedPointer(T* raw): pointer(raw) { }
    ScopedPointer(const ScopedPointer& other) = delete;
    ScopedPointer(ScopedPointer&& other) { … }
    ~ScopedPointer() { delete pointer; }
    ...
};

void foo() {
     ScopedPointer sp{new Triple{13, 42, 1}};
     if (...) {
          throw "ooops";
     }
     return sp->x + sp->z;
}
```

destructor is called, memory is freed => no leak

```cpp
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }
   ScopedPointer(const ScopedPointer& other) = delete;
   ScopedPointer(ScopedPointer&& other) { … }
   ~ScopedPointer() { delete pointer; }
   ...
};
```

```
template<typename T>
class ScopedPointer {
   T* pointer;
public:
   ScopedPointer(T* raw): pointer(raw) { }
   ~ScopedPointer() { delete pointer; }
   ...
};
```

```
template<typename T>
class ScopedPointer<T[]> {
   T* pointer;
public:
   ScopedPointer(T* raw): pointer(raw) { }
   ~ScopedPointer() { delete[] pointer; }
   ...
};
```

specialization for T[]

But what to do
with copy-paste?

```
template<typename T>
class ScopedPointer {
    T* pointer;
public:
    ScopedPointer(T* raw): pointer(raw) { }
    ~ScopedPointer() { delete pointer; }
    ...
};
```

```
template<typename T>
class ScopedPointer<T[]> {
    T* pointer;
public:
    ScopedPointer(T* raw): pointer(raw) { }
    ~ScopedPointer() { delete[] pointer; }
    ...
};
```

specialization for T[]

```cpp
template<typename T, typename Deleter = default_deleter<T>>
class ScopedPointer {
   T* pointer;
   Deleter deleter;

public:
   ScopedPointer(T* raw): pointer(raw) { }
   ~ScopedPointer() { deleter(pointer); }
   ...
};
```

```cpp
template<typename T> class default_deleter {
   void operator() (T* ptr) {
       delete ptr;
   }
};

template<typename T, typename Deleter = default_deleter<T>>
class ScopedPointer {
   T* pointer;
   Deleter deleter;

public:
   ScopedPointer(T* raw): pointer(raw) { }
   ~ScopedPointer() { deleter(pointer); }
   ...
};
```

```cpp
template<typename T> class default_deleter {
   void operator() (T* ptr) {
       delete ptr;
   }
};

template<typename T> class default_deleter<T[]> {
   void operator() (T* ptr) {
       delete[] ptr;
   }
};

template<typename T, typename Deleter = default_deleter<T>>
class ScopedPointer {
   T* pointer;
   Deleter deleter;

public:
   ScopedPointer(T* raw): pointer(raw) { }
   ~ScopedPointer() { deleter(pointer); }
};
```

```cpp
template<typename T> class default_deleter {
   void operator() (T* ptr) {
       delete ptr;
   }
};

template<typename T> class default_deleter<T[]> {
   void operator() (T* ptr) {
       delete[] ptr;
   }
};

template<typename T, typename Deleter = default_deleter<T>>
class ScopedPointer {
   T* pointer;
   Deleter deleter;

public:
   ScopedPointer(T* raw): pointer(raw) { }
   ~ScopedPointer() { deleter(pointer); }
};
```

Copy-paste is outlined into a separate class.

```cpp
template<typename T> class default_deleter {
   void operator() (T* ptr) {
       delete ptr;
   }
};

template<typename T> class default_deleter<T[]> {
   void operator() (T* ptr) {
       delete[] ptr;
   }
};

template<typename T, typename Deleter = default_deleter<T>>
class ScopedPointer {
   T* pointer;
   Deleter deleter;

public:
   ScopedPointer(T* raw): pointer(raw) { }
   ~ScopedPointer() { deleter(pointer); }
};
```

Copy-paste is outlined into a separate class.

It also means, that you can specify your own deleters!

deleters

```cpp
int foo() {

   auto fake_del = [](Triple* t){
       std::cout << "I love memory leaks!" << std::endl;
   };

   auto triples = new Triple[10];
   auto p = std::unique_ptr<Triple, decltype(fake_del)>(triples, fake_del);
   return 42;
}
```

deleters

It allows you to use unique_pointers not only for memory, but for any custom resource that need to be externally closed.

```
int foo() {

   auto fake_del = [](Triple* t){
       std::cout << "I love memory leaks!" << std::endl;
   };

   auto triples = new Triple[10];
   auto p = std::unique_ptr<Triple, decltype(fake_del)>(triples, fake_del);
   return 42;
}
```

```cpp
template <typename T>
class Tree {
   struct Node{
       Node* left, right;
       T data;
   };
   Node* root;

   void free_subtree(Node* node) {
       free_subtree(node->left);
       free_subtree(node->right);
       delete node;
   }
public:
   ~Tree() {
       free_subtree(root);
   }

   Node* find(T data) {
       return ...;
   }
};
```

```cpp
template <typename T>
class Tree {
   struct Node{
       Node* left, right;
       T data;
   };
   Node* root;

   void free_subtree(Node* node) {
       free_subtree(node->left);
       free_subtree(node->right);
       delete node;
   }
public:
   ~Tree() {
       free_subtree(root);
   }

   Node* find(T data) {
       return ...;
   }
};
```

A bit strange, but still valid implementation of binary tree.

But how to replace pointers with smart pointers here?

```cpp
template <typename T>
class Tree {
   struct Node{
       std::unique_ptr<Node> left, right;
       T data;
   };
   std::unique_ptr<Node> root;

   void free_subtree(Node* node) {
       free_subtree(node->left);
       free_subtree(node->right);
       delete node;
   }
public:
   ~Tree() {
       free_subtree(root);
   }

   Node* find(T data) {
       return ...;
   }
};
```

A bit strange, but still valid implementation of binary tree.

But how to replace pointers with smart pointers here?

```cpp
template <typename T>
class Tree {
   struct Node{
       std::unique_ptr<Node> left, right;
       T data;
   };
   std::unique_ptr<Node> root;

   void free_subtree(std::unique_ptr<Node> node) {
       free_subtree(node->left);
       free_subtree(node->right);
       delete node;
   }
public:
   ~Tree() {
       free_subtree(root);
   }

   Node* find(T data) {
       return ...;
   }
};
```

A bit strange, but still valid implementation of binary tree.

But how to replace pointers with smart pointers here?

```cpp
template <typename T>
class Tree {
   struct Node{
       std::unique_ptr<Node> left, right;
       T data;
   };
   std::unique_ptr<Node> root;

   void free_subtree(std::unique_ptr<Node> node) {
       // nothing to do!
       // everything will be done in destructor!
   }

public:
   ~Tree() {
       free_subtree(root);
   }

   Node* find(T data) {
       return ...;
   }
};
```

A bit strange, but still valid implementation of binary tree.

But how to replace pointers with smart pointers here?

```cpp
template <typename T>
class Tree {
   struct Node{
       std::unique_ptr<Node> left, right;
       T data;
   };
   std::unique_ptr<Node> root;

   void free_subtree(std::unique_ptr<Node> node) {
       // nothing to do!
       // everything will be done in destructor!
   }

public:
   ~Tree() {
       free_subtree(std::move(root));
   }

   Node* find(T data) {
       return ...;
   }
};
```

A bit strange, but still valid implementation of binary tree.

But how to replace pointers with smart pointers here?

```cpp
template <typename T>
class Tree {
   struct Node{
       std::unique_ptr<Node> left, right;
       T data;
   };
   std::unique_ptr<Node> root;

   void free_subtree(std::unique_ptr<Node> node) {
       // nothing to do!
       // everything will be done in destructor!
   }

public:
   ~Tree() {
       free_subtree(std::move(root));
   }

   Node* find(T data) {
       return ...;
   }
};
```

A bit strange, but still valid implementation of binary tree.

But how to replace pointers with smart pointers here?

But what should we return here?

```cpp
template <typename T>
class Tree {
   struct Node{
       std::unique_ptr<Node> left, right;
       T data;
   };
   std::unique_ptr<Node> root;

   void free_subtree(std::unique_ptr<Node> node) {
       // nothing to do!
       // everything will be done in destructor!
   }

public:
   ~Tree() {
       free_subtree(std::move(root));
   }

   Node* find(T data) {
       return ...;
   }
};
```

A bit strange, but still valid implementation of binary tree.

But how to replace pointers with smart pointers here?

But what should we return here?
Transferring ownership will be strange.
Raw pointer or reference => dangerous.

```cpp
template <typename T>
class Tree {
   struct Node{
       std::unique_ptr<Node> left, right;
       T data;
   };
   std::unique_ptr<Node> root;

   void free_subtree(std::unique_ptr<Node> node) {
       // nothing to do!
       // everything will be done in destructor!
   }

public:
   ~Tree() {
       free_subtree(std::move(root));
   }

   Node* find(T data) {
       return ...;
   }
};
```

A bit strange, but still valid implementation of binary tree.

But how to replace pointers with smart pointers here?

We need to somehow share ownership!

But what should we return here?
Transferring ownership will be strange.
Raw pointer or reference => dangerous.

shared pointers

```
int foo() {
    auto triple = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(triple);



    return 42;
}
```

shared pointers

```
int foo() {
    auto triple = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(triple);
```

New shared pointer created.

```
    return 42;
}
```

shared pointers

```
int foo() {
    auto triple = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(triple);




    return 42;
}
```

New shared pointer created.

Previously there must be no other shared pointers on that object!

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto triple = new Triple{1, 2, 3};
   auto p = std::shared_ptr<Triple>(triple);


   print(p);


   return 42;
}
```

New shared pointer created.

Previously there must be no other shared pointers on that object!

shared pointers

created this copy via
copy constructor

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto triple = new Triple{1, 2, 3};
   auto p = std::shared_ptr<Triple>(triple);


   print(p);


   return 42;
}
```

New shared pointer
created.

Previously there must
be no other shared
pointers on that
object!

shared pointers

created this copy via copy constructor and shared the ownership with it!

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto triple = new Triple{1, 2, 3};
   auto p = std::shared_ptr<Triple>(triple);


   print(p);


   return 42;
}
```

New shared pointer created.

Previously there must be no other shared pointers on that object!

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto triple = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(triple);

    print(p);

    return 42;
}
```

created this copy via copy constructor and shared the ownership with it!

Now 2 shared ptrs owns the resource.

New shared pointer created.

Previously there must be no other shared pointers on that object!

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto triple = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(triple);


    print(p);


    return 42;
}
```

created this copy via copy constructor and shared the ownership with it!

Now 2 shared ptrs owns the resource.

Resource is freed when no one owns it.

New shared pointer created.

Previously there must be no other shared pointers on that object!

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto triple = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(triple);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```

created this copy via copy constructor and shared the ownership with it!

Now 2 shared ptrs owns the resource.

Resource is freed when no one owns it.

New shared pointer created.

Previously there must be no other shared pointers on that object!

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto triple = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(triple);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```

Doesn’t it look familiar to you?

This is absolutely reference counting memory management!

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto triple = new Triple{1, 2, 3};
   auto p = std::shared_ptr<Triple>(triple);

   std::shared_ptr<Triple> p2 = p;
   print(p);
   print(p2);
   p = nullptr;
   return 42;
}
```

Reference counter: 0

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto triple = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(triple);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```

Reference counter: 1

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto triple = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(triple);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```

Reference counter: 2

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto triple = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(triple);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```

Reference counter: 2

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto triple = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(triple);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```

Reference counter: 3

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto triple = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(triple);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```

Reference counter: 2

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto triple = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(triple);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```

Reference counter: 2

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto triple = new Triple{1, 2, 3};
   auto p = std::shared_ptr<Triple>(triple);

   std::shared_ptr<Triple> p2 = p;
   print(p);
   print(p2);
   p = nullptr;
   return 42;
}
```

Reference counter: 3

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto triple = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(triple);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```

Reference counter: 2

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto triple = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(triple);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```

Reference counter: 2

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto triple = new Triple{1, 2, 3};
   auto p = std::shared_ptr<Triple>(triple);

   std::shared_ptr<Triple> p2 = p;
   print(p);
   print(p2);
   p = nullptr;
   return 42;
}
```

Reference counter: 1

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto triple = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(triple);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```

Reference counter: 0

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto triple = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(triple);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```

Reference counter: 0

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto triple = new Triple{1, 2, 3};
   auto p = std::shared_ptr<Triple>(triple);

   std::shared_ptr<Triple> p2 = p;
   print(p);
   print(p2);
   p = nullptr;
   return 42;        delete triple;
}
```

Memory is deleted as no one owns it.

```cpp
template <typename T>
class Tree {
   struct Node{
       std::unique_ptr<Node> left, right;
       T data;
   };
   std::unique_ptr<Node> root;

   void free_subtree(std::unique_ptr<Node> node) {
       // nothing to do!
       // everything will be done in destructor!
   }

public:
   ~Tree() {
       free_subtree(std::move(root));
   }

   Node* find(T data) {
       return ...;
   }
};
```

A bit strange, but still valid implementation of binary tree.

But how to replace pointers with smart pointers here?

We need to somehow share ownership!

But what should we return here?
Transferring ownership will be strange.
Raw pointer or reference => dangerous.

```cpp
template <typename T>
class Tree {
   struct Node{
       std::shared_ptr<Node> left, right;
       T data;
   };
   std::shared_ptr<Node> root;

public:

   std::shared_ptr<Node> find(T data) {
       return ...;
   }
};
```

A bit strange, but still valid implementation of binary tree.

But how to replace pointers with smart pointers here?

We need to somehow share ownership!

But what should we return here?
Transferring ownership will be strange.
Raw pointer or reference => dangerous.

Reference counter: 0

shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto t = new Triple{1, 2, 3};
   auto p = std::shared_ptr<Triple>(t);

   std::shared_ptr<Triple> p2 = p;
   print(p);
   print(p2);
   p = nullptr;
   return 42;
}
```

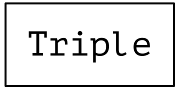
```
delete triple;
```

How to implement such functionality?

Memory is deleted as no one owns it.

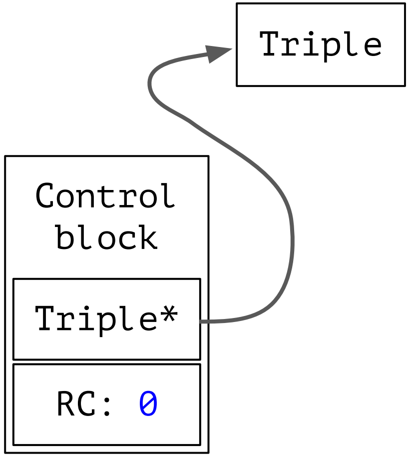
shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto t = new Triple{1, 2, 3};
   auto p = std::shared_ptr<Triple>(t);

   std::shared_ptr<Triple> p2 = p;
   print(p);
   print(p2);
   p = nullptr;
   return 42;
}
```



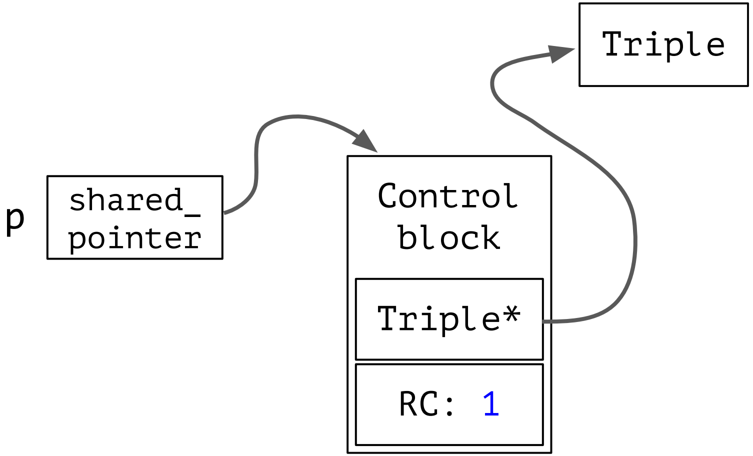
shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto t = new Triple{1, 2, 3};
   auto p = std::shared_ptr<Triple>(t);

   std::shared_ptr<Triple> p2 = p;
   print(p);
   print(p2);
   p = nullptr;
   return 42;
}
```



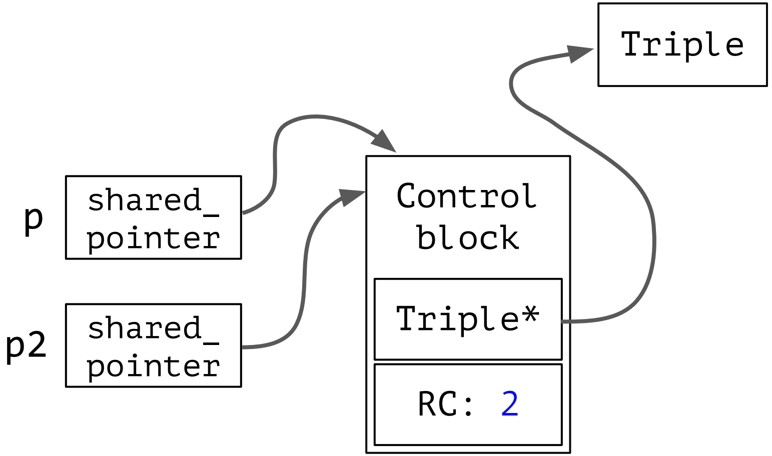
shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto t = new Triple{1, 2, 3};
   auto p = std::shared_ptr<Triple>(t);

   std::shared_ptr<Triple> p2 = p;
   print(p);
   print(p2);
   p = nullptr;
   return 42;
}
```



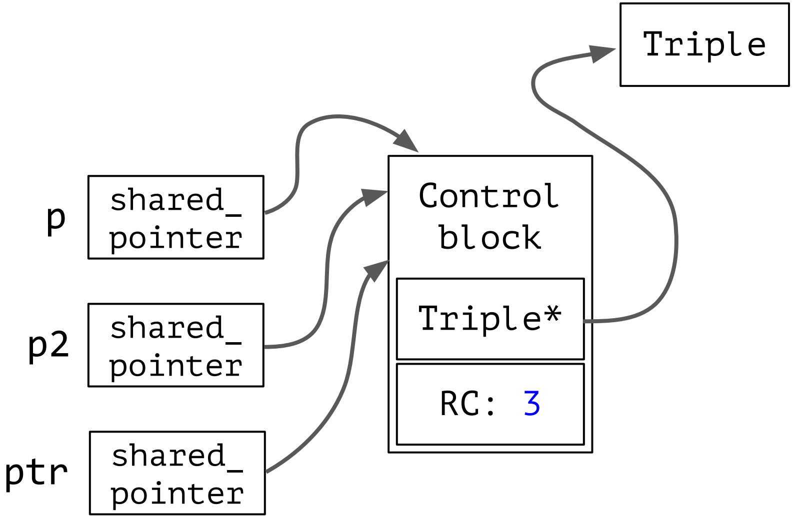
shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto t = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(t);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```



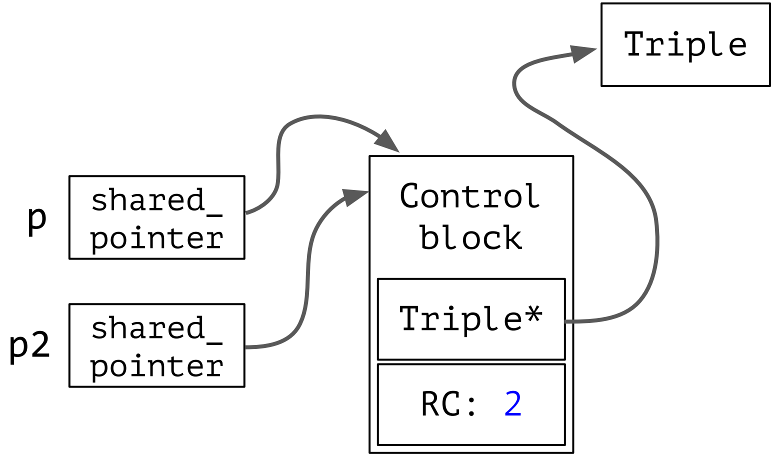
shared pointers

```cpp
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto t = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(t);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```



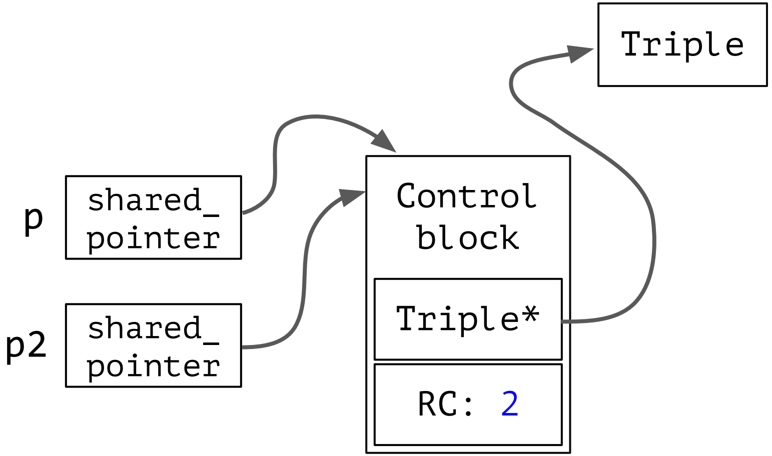
shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto t = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(t);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```



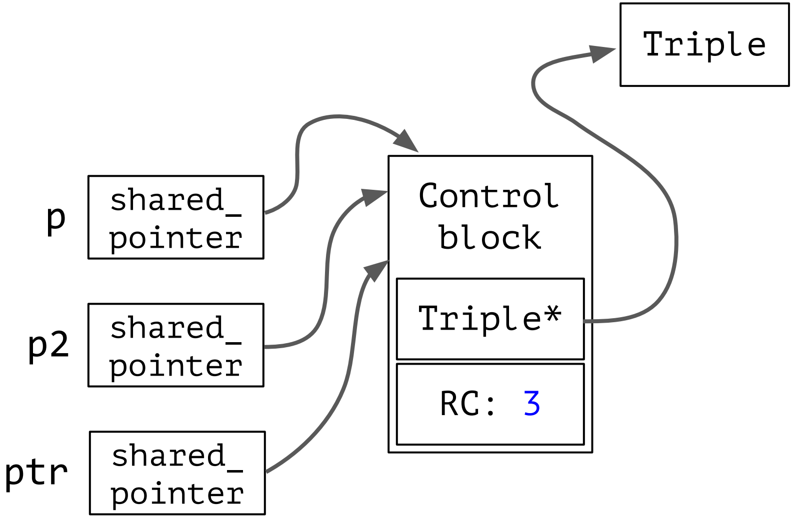
shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto t = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(t);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```



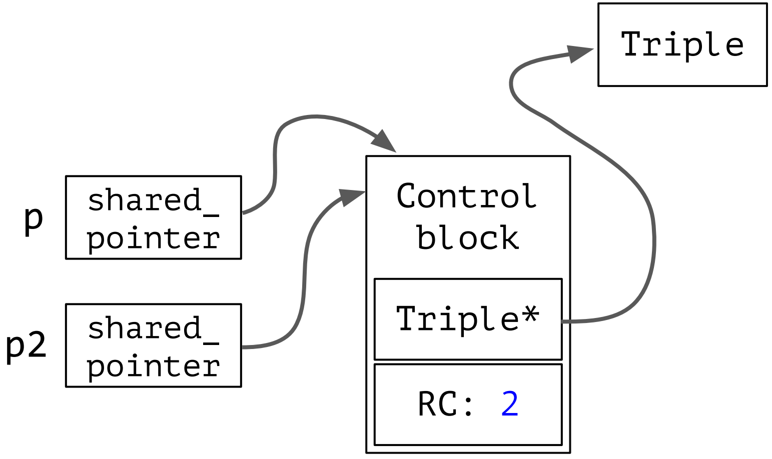
shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto t = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(t);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```



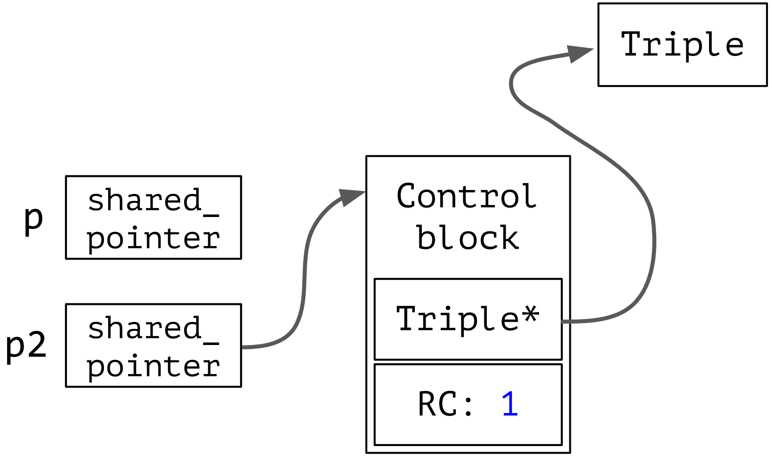
shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto t = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(t);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```



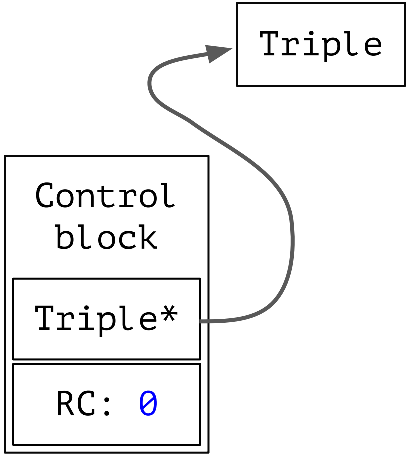
shared pointers

```cpp
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto t = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(t);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```



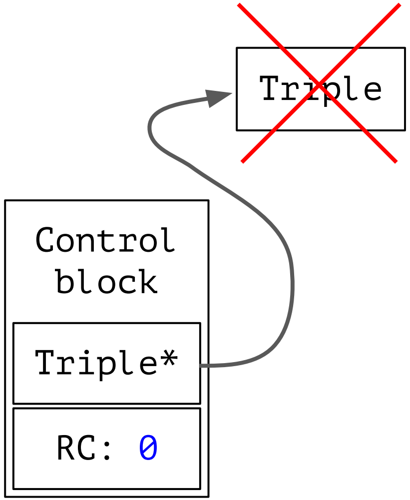
shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto t = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(t);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```



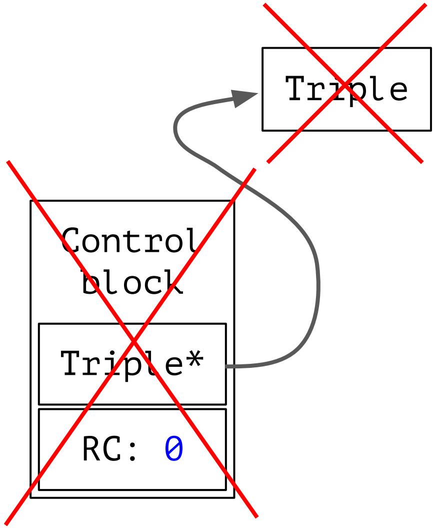
shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto t = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(t);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```



shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto t = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(t);

    std::shared_ptr<Triple> p2 = p;
    print(p);
    print(p2);
    p = nullptr;
    return 42;
}
```

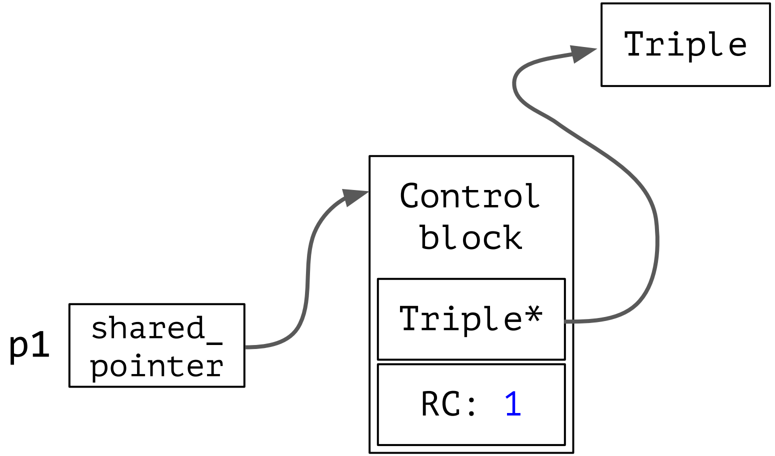


shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto t = new Triple{1, 2, 3};
   auto p1 = std::shared_ptr<Triple>(t);
   auto p2 = std::shared_ptr<Triple>(t);
   return 42;
}
```

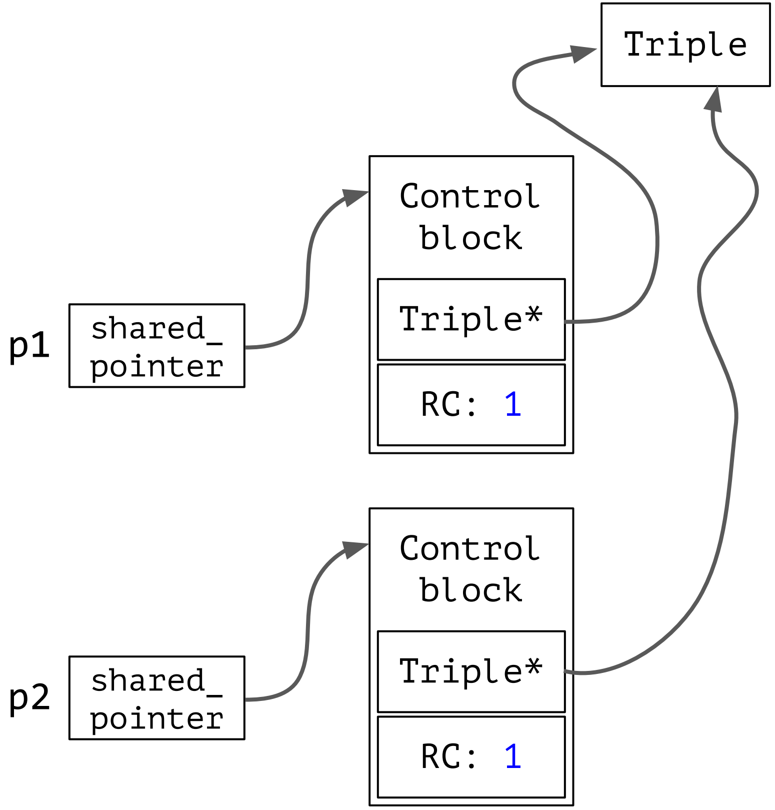
shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto t = new Triple{1, 2, 3};
    auto p1 = std::shared_ptr<Triple>(t);
    auto p2 = std::shared_ptr<Triple>(t);
    return 42;
}
```



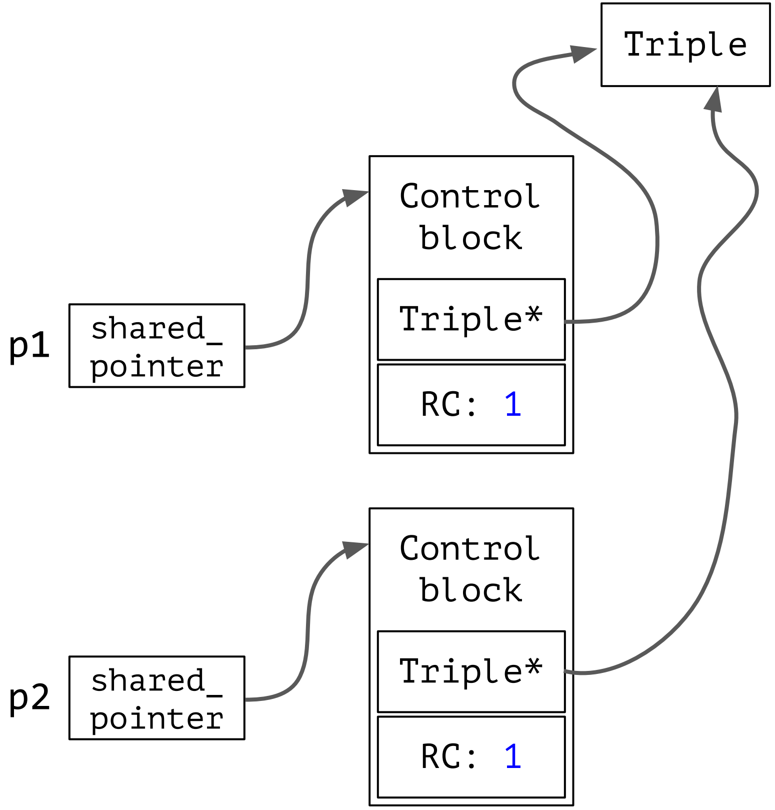
shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto t = new Triple{1, 2, 3};
   auto p1 = std::shared_ptr<Triple>(t);
   auto p2 = std::shared_ptr<Triple>(t);
   return 42;
}
```



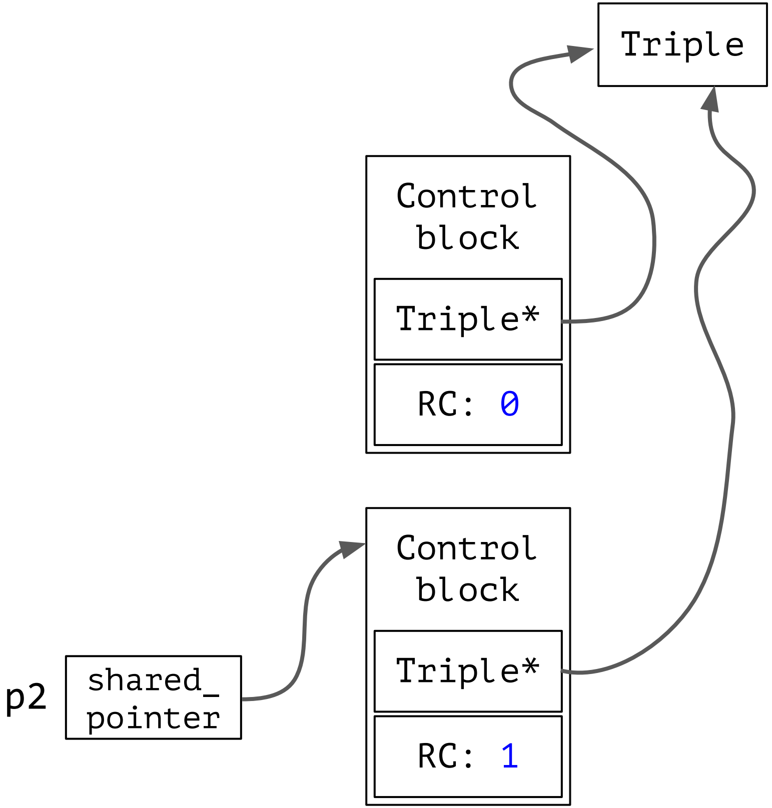
shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto t = new Triple{1, 2, 3};
   auto p1 = std::shared_ptr<Triple>(t);
   auto p2 = std::shared_ptr<Triple>(t);
   return 42;
}
```



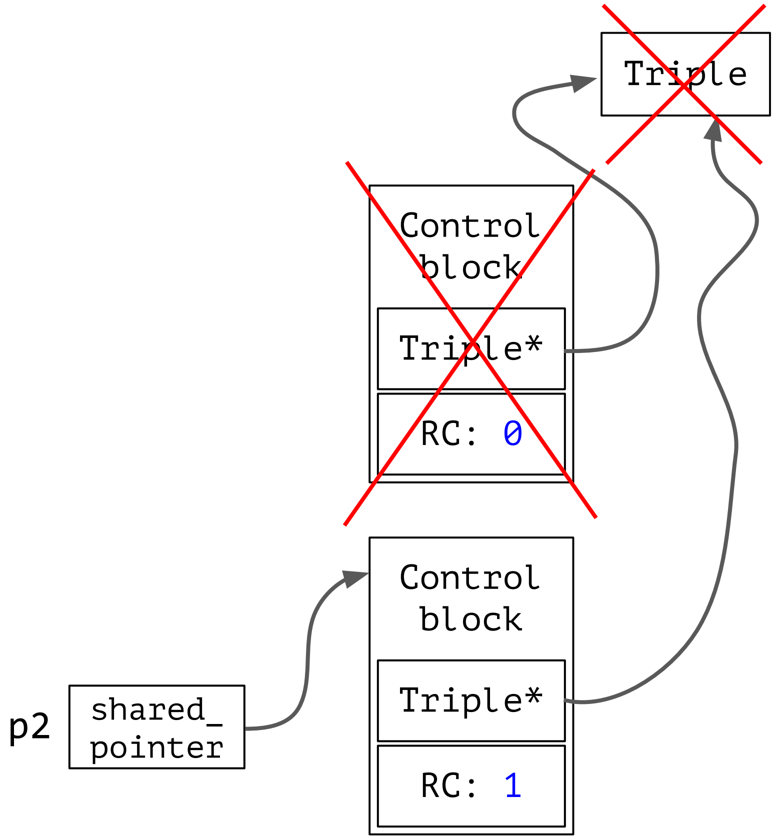
shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto t = new Triple{1, 2, 3};
    auto p1 = std::shared_ptr<Triple>(t);
    auto p2 = std::shared_ptr<Triple>(t);
    return 42;
}
```



shared pointers

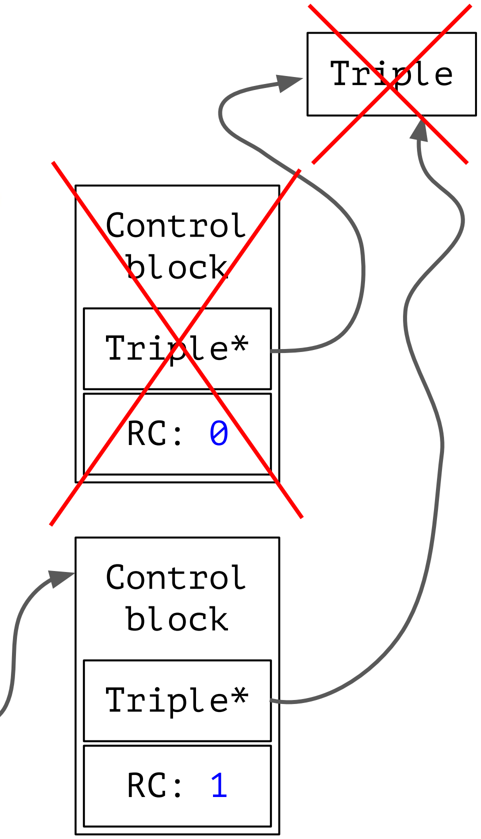
```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto t = new Triple{1, 2, 3};
   auto p1 = std::shared_ptr<Triple>(t);
   auto p2 = std::shared_ptr<Triple>(t);
   return 42;
}
```



shared pointers

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto t = new Triple{1, 2, 3};
    auto p1 = std::shared_ptr<Triple>(t);
    auto p2 = std::shared_ptr<Triple>(t);
    return 42;
}
```

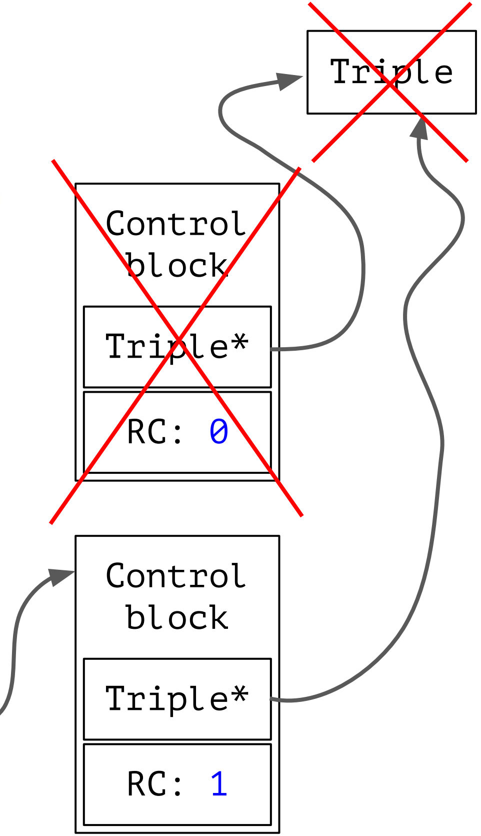
double free here again.



shared pointers

```cpp
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto t = new Triple{1, 2, 3};
    auto p1 = std::shared_ptr<Triple>(t);
    auto p2 = std::shared_ptr<Triple>(t);
    return 42;
}
```

double free here again.



shared pointers from this

What do you think about such code?

shared pointers from this

```
struct Foo {
   std::shared_ptr<Foo> giveMeSharedPointer() {
       return std::shared_ptr<Foo>(this);
   }
};


int main() {
   Foo* f = new Foo();
   std::shared_ptr<Foo> ptr1(f);
   std::shared_ptr<Foo> ptr2 = f->giveMeSharedPointer();
   return 0;
}
```

What do you think about such code?

shared pointers from this

```
struct Foo {
    std::shared_ptr<Foo> giveMeSharedPointer() {
        return std::shared_ptr<Foo>(this);
    }
};


int main() {
    Foo* f = new Foo();
    std::shared_ptr<Foo> ptr1(f);
    std::shared_ptr<Foo> ptr2 = f->giveMeSharedPointer();
    return 0;
}
```

double free again
(absolutely same reasons!)

shared pointers from this

```
class Foo: public std::enable_shared_from_this<Foo> {
    std::shared_ptr<Foo> giveMeSharedPointer() {
        return shared_from_this();
    }
};


int main() {
   Foo* f = new Foo();
   std::shared_ptr<Foo> ptr1(f);
   std::shared_ptr<Foo> ptr2 = f->giveMeSharedPointer();
   return 0;
}
```

shared pointers from this

```
class Foo: public std::enable_shared_from_this<Foo> {
    std::shared_ptr<Foo> giveMeSharedPointer() {
        return shared_from_this();
    }
};


int main() {
   Foo* f = new Foo();
   std::shared_ptr<Foo> ptr1(f);
   std::shared_ptr<Foo> ptr2 = f->giveMeSharedPointer();
   return 0;
}
```

shared pointers from this

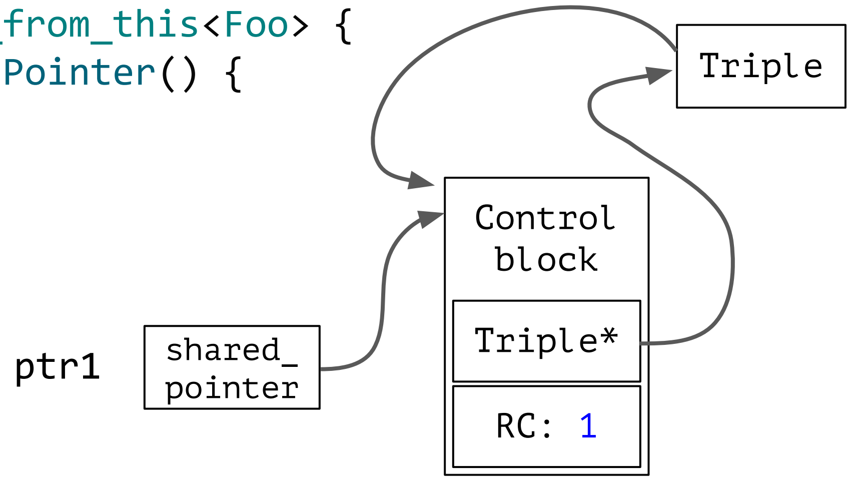
How does it called?
CRTP! We are adding special fields (pointer to controller block) to each instance of such classes.

```
class Foo: public std::enable_shared_from_this<Foo> {
    std::shared_ptr<Foo> giveMeSharedPointer() {
        return shared_from_this();
    }
};


int main() {
   Foo* f = new Foo();
   std::shared_ptr<Foo> ptr1(f);
   std::shared_ptr<Foo> ptr2 = f->giveMeSharedPointer();
   return 0;
}
```

shared pointers from this

How does it called?
CRTP! We are adding special fields (pointer to controller block) to each instance of such classes.

```
class Foo: public std::enable_shared_from_this<Foo> {
    std::shared_ptr<Foo> giveMeSharedPointer() {
        return shared_from_this();
    }
};


int main() {
   Foo* f = new Foo();
   std::shared_ptr<Foo> ptr1(f);
   std::shared_ptr<Foo> ptr2 = f->giveMeSharedPointer();
   return 0;
}
```



shared pointers from this

How does it called?
CRTP! We are adding special fields (pointer to controller block) to each instance of such classes.

```
class Foo: public std::enable_shared_from_this<Foo> {
    std::shared_ptr<Foo> giveMeSharedPointer() {
        return shared_from_this();
    }
};


int main() {
   Foo* f = new Foo();
   std::shared_ptr<Foo> ptr1(f);
   std::shared_ptr<Foo> ptr2 =
                 f->giveMeSharedPointer();
   return 0;
}
```

Limitations:

1. If shared_from_this is called before CB was created => UB

shared pointers from this

How does it called?
CRTP! We are adding special fields (pointer to controller block) to each instance of such classes.

```
class Foo: public std::enable_shared_from_this<Foo> {
    std::shared_ptr<Foo> giveMeSharedPointer() {
        return shared_from_this();
    }
};


int main() {
   Foo* f = new Foo();
   std::shared_ptr<Foo> ptr1(f);
   std::shared_ptr<Foo> ptr2 =
                f->giveMeSharedPointer();
   return 0;
}
```

Limitations:

1. If shared_from_this is called before CB was created => UB

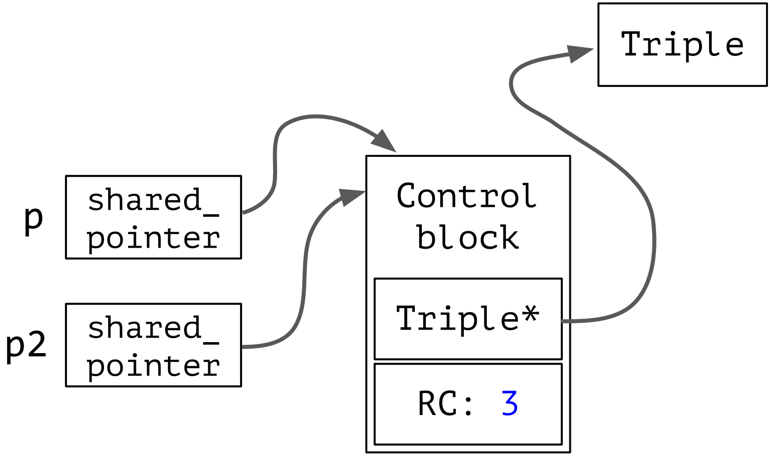
2. No guarantees when calling from ctr,

3. UB guaranteed when calling from dtr

make_shared

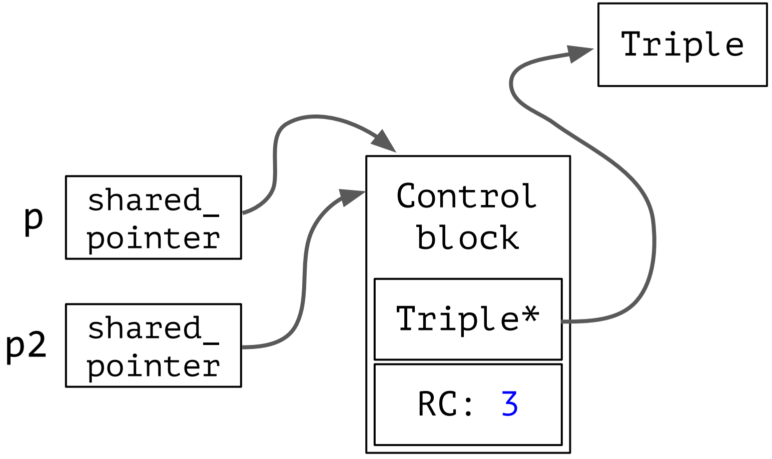
make_shared

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto t = new Triple{1, 2, 3};
   auto p = std::shared_ptr<Triple>(t);

   std::shared_ptr<Triple> p2 = p;
   print(p);
   print(p2);
   p = nullptr;
   return 42;
}
```



make_shared

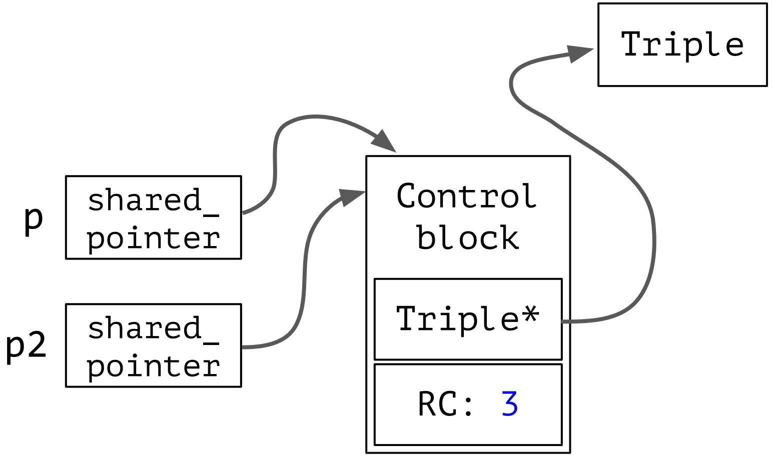
```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto p =
     std::make_shared<Triple>(1, 2, 3);

   std::shared_ptr<Triple> p2 = p;
   print(p);
   print(p2);
   p = nullptr;
   return 42;
}
```



make_shared

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto p =
     std::make_shared<Triple>(1, 2, 3);

   std::shared_ptr<Triple> p2 = p;
   print(p);
   print(p2);
   p = nullptr;
   return 42;
}
```

How do you think will this picture change?

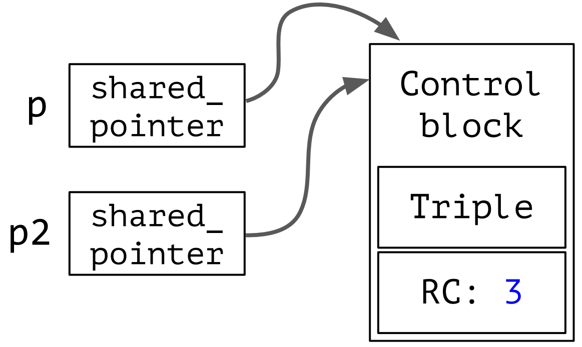


make_shared

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto p =
     std::make_shared<Triple>(1, 2, 3);

   std::shared_ptr<Triple> p2 = p;
   print(p);
   print(p2);
   p = nullptr;
   return 42;
}
```

How do you think will
this picture change?

Yes! We can "inline" data
into control block.

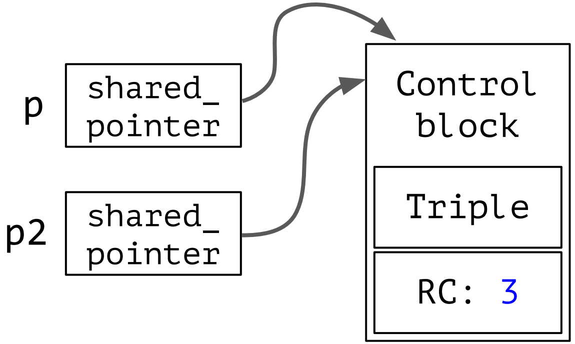


make_shared

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto p =
     std::make_shared<Triple>(1, 2, 3);

   std::shared_ptr<Triple> p2 = p;
   print(p);
   print(p2);
   p = nullptr;
   return 42;
}
```

How do you think will this picture change?

Yes! We can "inline" data into control block.



Cool, but has some drawbacks, will discuss later.

What is the main problem of reference counting?

What is the main problem of reference counting?

Circular references and Circular garbage!

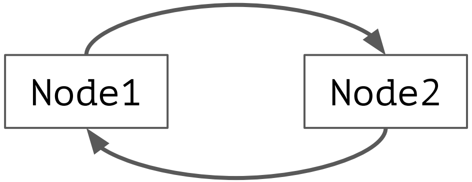
```cpp
template <typename T>
struct Node {
    std::shared_ptr<Node> prev;
    std::shared_ptr<Node> next;

    ~Node() {
        std::cout << "node deleted" << std::endl;
    }
};
```

```cpp
template <typename T>
struct Node {
    std::shared_ptr<Node> prev;
    std::shared_ptr<Node> next;

    ~Node() {
        std::cout << "node deleted" << std::endl;
    }
};

int main() {

    {
        auto n1 = std::make_shared<Node<int>>();
        auto n2 = std::make_shared<Node<int>>();
        n1->next = n2;
        n2->prev = n1;

    }
    ...
}
```

```
template <typename T>
struct Node {
    std::shared_ptr<Node> prev;
    std::shared_ptr<Node> next;

    ~Node() {
        std::cout << "node deleted" << std::endl;
    }
};

int main() {

    {
        auto n1 = std::make_shared<Node<int>>();
        auto n2 = std::make_shared<Node<int>>();
        n1->next = n2;
        n2->prev = n1;
```

MEMORY LEAK!

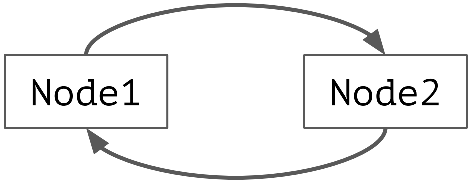
```
    }
    ...
}
```



```cpp
template <typename T>
struct Node {
    std::shared_ptr<Node> prev;
    std::shared_ptr<Node> next;

    ~Node() {
        std::cout << "node deleted" << std::endl;
    }
};

int main() {

    {
        auto n1 = std::make_shared<Node<int>>();
        auto n2 = std::make_shared<Node<int>>();
        n1->next = n2;
        n2->prev = n1;
    }
    ...
}
```

MEMORY LEAK!

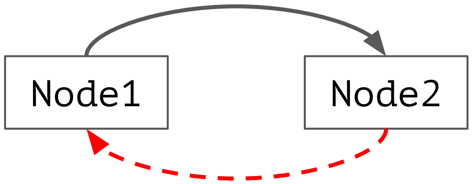
The law of nature: pure reference counting => weak references



```cpp
template <typename T>
struct Node {
    std::weak_ptr<Node> prev;
    std::shared_ptr<Node> next;

    ~Node() {
        std::cout << "node deleted" << std::endl;
    }
};

int main() {

    {
        auto n1 = std::make_shared<Node<int>>();
        auto n2 = std::make_shared<Node<int>>();
        n1->next = n2;
        n2->prev = n1;
    }
    ...
}
```

~~MEMORY LEAK!~~

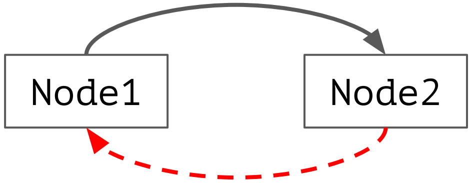
The law of nature: pure reference counting => weak references (pointers in our case)



```cpp
template <typename T>
struct Node {
    std::weak_ptr<Node> prev;
    std::shared_ptr<Node> next;

    ~Node() {
        std::cout << "node deleted" << std::endl;
    }
};

int main() {

    {
        auto n1 = std::make_shared<Node<int>>();
        auto n2 = std::make_shared<Node<int>>();
        n1->next = n2;
        n2->prev = n1;
    }
    ...
}
```

```
node deleted
node deleted
```

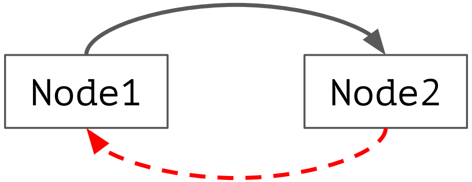
The law of nature: pure reference counting => weak references (pointers in our case)



```cpp
template <typename T>
struct Node {
    std::weak_ptr<Node> prev;
    std::shared_ptr<Node> next;

    ~Node() {
        std::cout << "node deleted" << std::endl;
    }
};

int main() {

    {
        auto n1 = std::make_shared<Node<int>>();
        auto n2 = std::make_shared<Node<int>>();
        n1->next = n2;
        n2->prev = n1;
    }
    ...
}
```

```
node deleted
node deleted
```

But why and how to use such stuff?



But why and how to use such stuff?

```
template <typename T>
struct Node {
   std::weak_ptr<Node> prev;
   std::shared_ptr<Node> next;

   void print_previous_one() {
      auto strong_pointer = prev.lock();
      if (strong_pointer) {
         std::cout << "previous element is " << *strong_pointer << std::endl;
      } else {
         std::cout << "well, no luck, previous is already expired" << std::endl;
      }
   }
};


int main() {
   ...
}
```

But why and how to use such stuff?

```
template <typename T>
struct Node {
    std::weak_ptr<Node> prev;
    std::shared_ptr<Node> next;

    void print_previous_one() {
        auto strong_pointer = prev.lock();
        if (strong_pointer) {
            std::cout << "previous element is " << *strong_pointer << std::endl;
        } else {
            std::cout << "well, no luck, previous is already expired" << std::endl;
        }
    }
};


int main() {
    ...
}
```

if object is still alive, you'll take a shared_ptr to it

```cpp
template <typename T>
struct Node {
    std::weak_ptr<Node> prev;
    std::shared_ptr<Node> next;

    void print_previous_one() {
        auto strong_pointer = prev.lock();
        if (strong_pointer) {
            std::cout << "previous element is " << *strong_pointer << std::endl;
        } else {
            std::cout << "well, no luck, previous is already expired" << std::endl;
        }
    }
};


int main() {
    ...
}
```

But why and how to use such stuff?

if object is still alive, you’ll take a shared_ptr to it

otherwise, you’ll just have an empty shared_pointer

```
template <typename T>
struct Node {
    std::weak_ptr<Node> prev;
    std::shared_ptr<Node> next;

    void print_previous_one() {
        auto strong_pointer = prev.lock();
        if (strong_pointer) {
            std::cout << "previous element is " << *strong_pointer << std::endl;
        } else {
            std::cout << "well, no luck, previous is already expired" << std::endl;
        }
    }
};

int main() {
    ...
}
```

But why and how to use such stuff?

if object is still alive, you'll take a shared_ptr to it

otherwise, you'll just have an empty shared_pointer

Why is it better than just using raw pointer?

But why and how to use such stuff?

```
template <typename T>
struct Node {
    std::weak_ptr<Node> prev;
    std::shared_ptr<Node> next;

    void print_previous_one() {
        auto strong_pointer = prev.lock();
        if (strong_pointer) {
            std::cout << "previous element is " << *strong_pointer << std::endl;
        } else {
            std::cout << "well, no luck, previous is already expired" << std::endl;
        }
    }
};
```

if object is still alive, you’ll take a shared_ptr to it

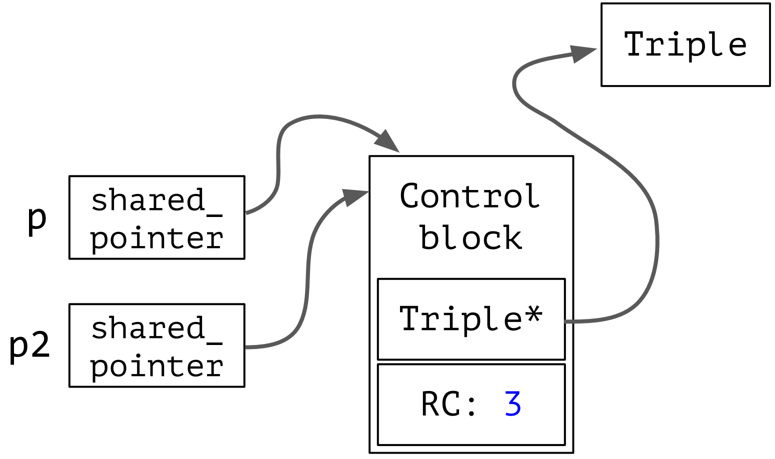
otherwise, you’ll just have an empty shared_pointer

```
int main() {
    ...
}
```

Why is it better than just using raw pointer? Because raw pointer can be dangling pointer, you can’t control that. Here you either have correct pointer to object or nothing.

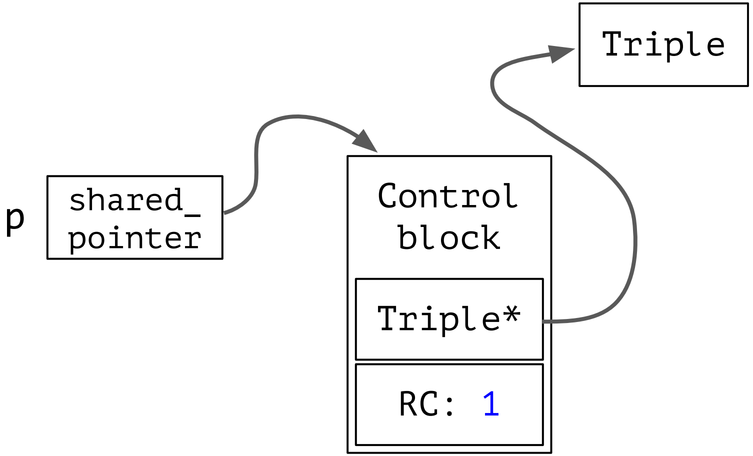
make_shared

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   Triple* t = new Triple{1, 2, 3};
   auto p = std::shared_ptr<Triple>(t);

   std::shared_ptr<Triple> p2 = p;
   print(p);
   print(p2);
   p = nullptr;
   return 42;
}
```



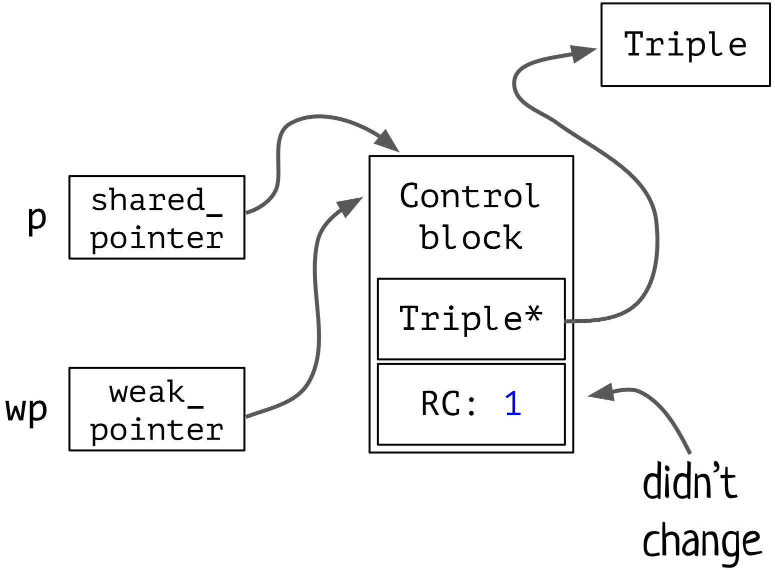
make_shared

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   Triple* t = new Triple{1, 2, 3};
   auto p = std::shared_ptr<Triple>(t);
   std::weak_ptr<Triple> wp = p;

   print(p);
   p = nullptr;
   return 42;
}
```



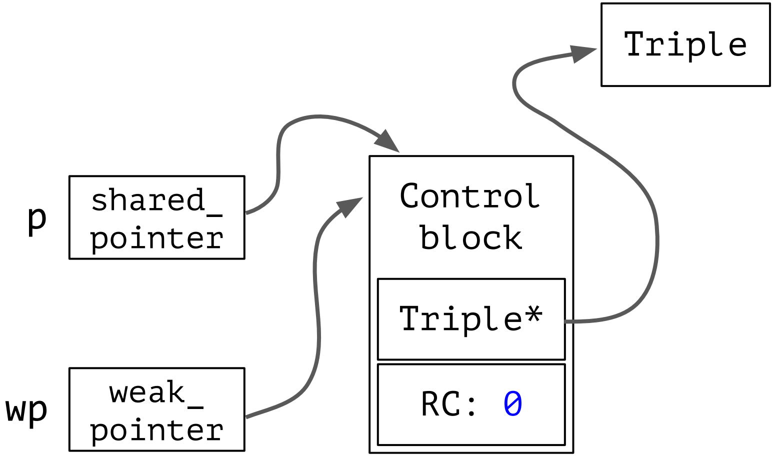
make_shared

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    Triple* t = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(t);
    std::weak_ptr<Triple> wp = p;

    print(p);
    p = nullptr;
    return 42;
}
```



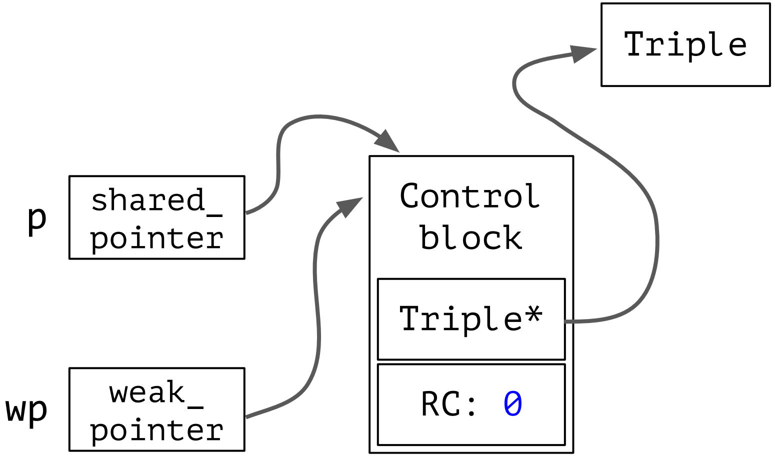
make_shared

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   Triple* t = new Triple{1, 2, 3};
   auto p = std::shared_ptr<Triple>(t);
   std::weak_ptr<Triple> wp = p;

   print(p);
   p = nullptr;
   return 42;
}
```



make_shared

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    Triple* t = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(t);
    std::weak_ptr<Triple> wp = p;

    print(p);
    p = nullptr;
    return 42;
}
```

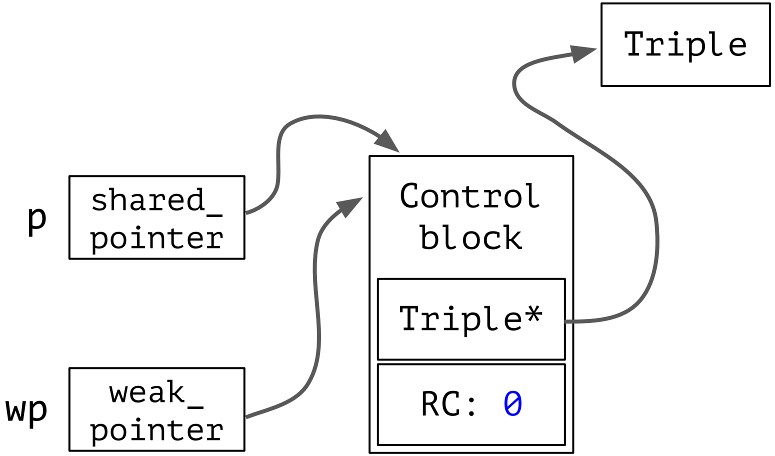


Can't we deallocate object?

Can we deallocate control block?

make_shared

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    Triple* t = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(t);
    std::weak_ptr<Triple> wp = p;

    print(p);
    p = nullptr;
    return 42;
}
```

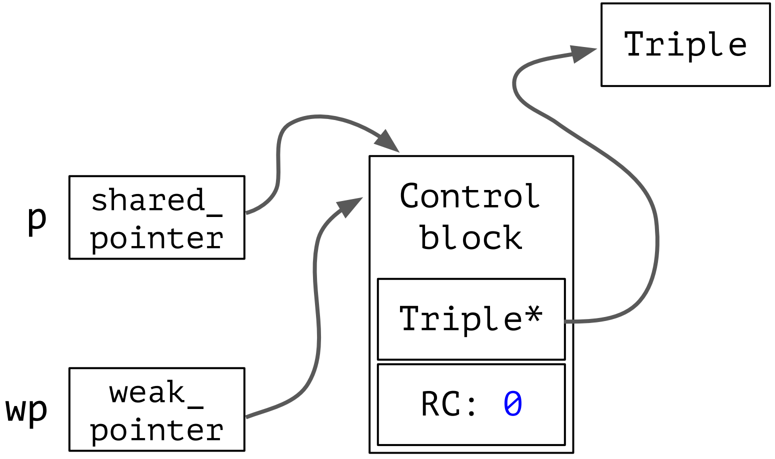


Can't we deallocate object? YES,

Can we deallocate control block? NO,

make_shared

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    Triple* t = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(t);
    std::weak_ptr<Triple> wp = p;

    print(p);
    p = nullptr;
    return 42;
}
```

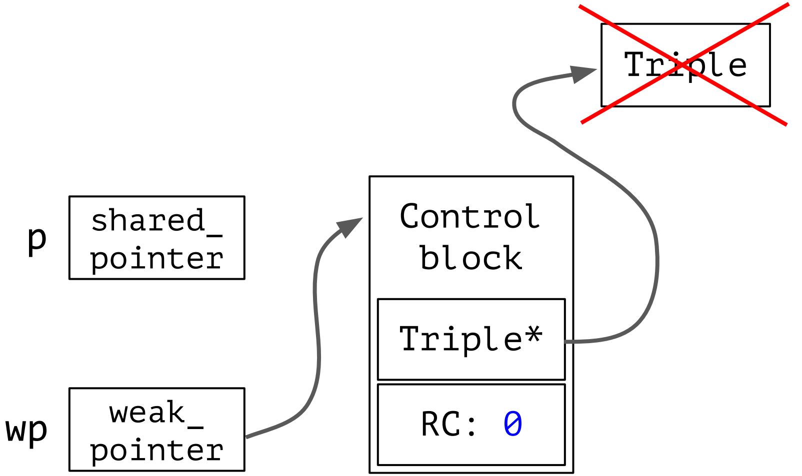


Can't we deallocate object? YES,

Can we deallocate control block? NO,
or incorrect weak_ptr will appear.

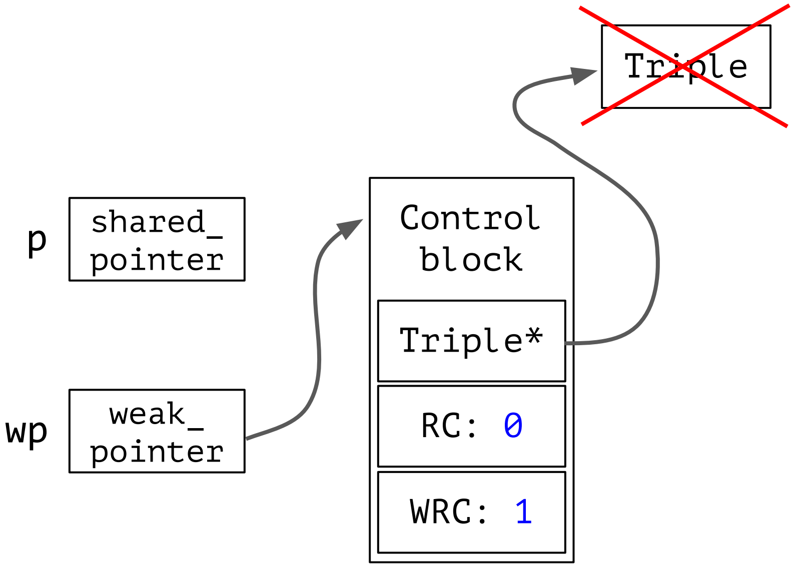
make_shared

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    Triple* t = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(t);
    std::weak_ptr<Triple> wp = p;

    print(p);
    p = nullptr;
    return 42;
}
```



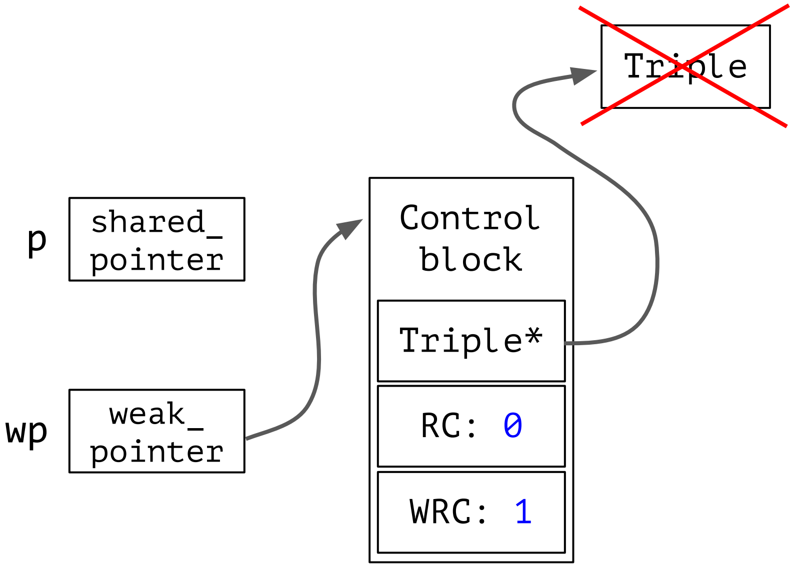
make_shared

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   Triple* t = new Triple{1, 2, 3};
   auto p = std::shared_ptr<Triple>(t);
   std::weak_ptr<Triple> wp = p;

   print(p);
   p = nullptr;
   return 42;
}
```



make_shared

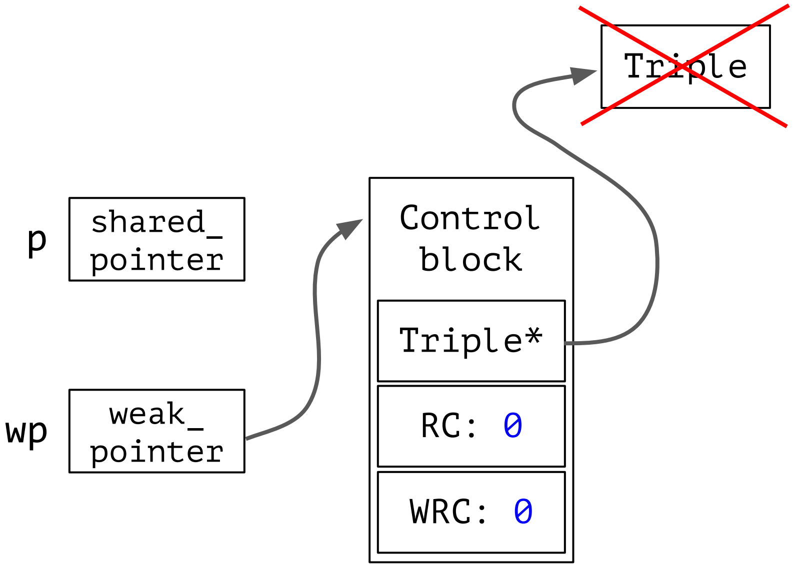
```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   Triple* t = new Triple{1, 2, 3};
   auto p = std::shared_ptr<Triple>(t);
   std::weak_ptr<Triple> wp = p;

   print(p);
   p = nullptr;
   return 42;
}
```



When there are also no weak_ptrs to this control block => we can deallocate it as well.

make_shared

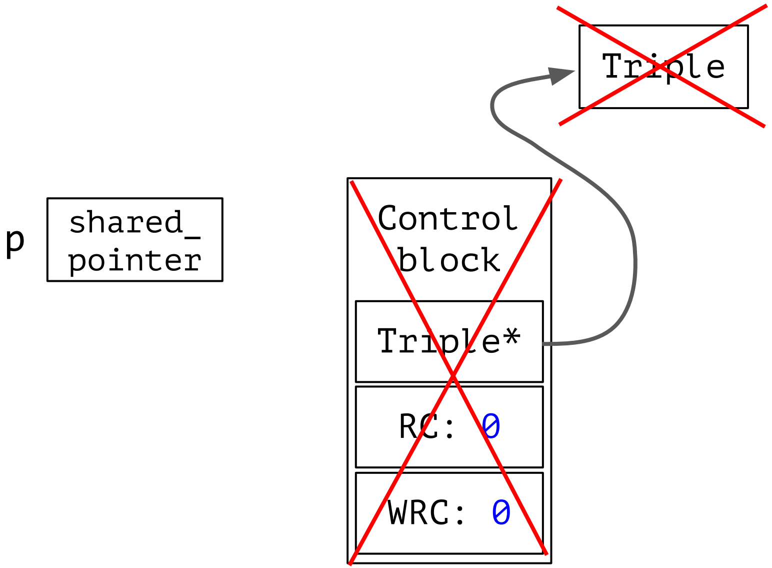
```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    Triple* t = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(t);
    std::weak_ptr<Triple> wp = p;

    print(p);
    p = nullptr;
    return 42;
}
```



When there are also no weak_ptrs to this control block => we can deallocate it as well.

make_shared

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    Triple* t = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(t);
    std::weak_ptr<Triple> wp = p;

    print(p);
    p = nullptr;
    return 42;
}
```

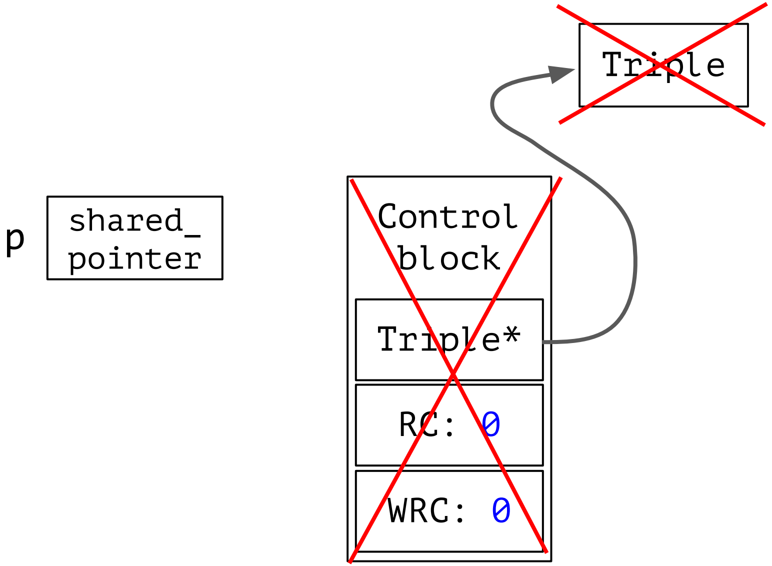


When there are also no weak_ptrs to this control block => we can deallocate it as well.

It can be important for make_shared case!

make_shared

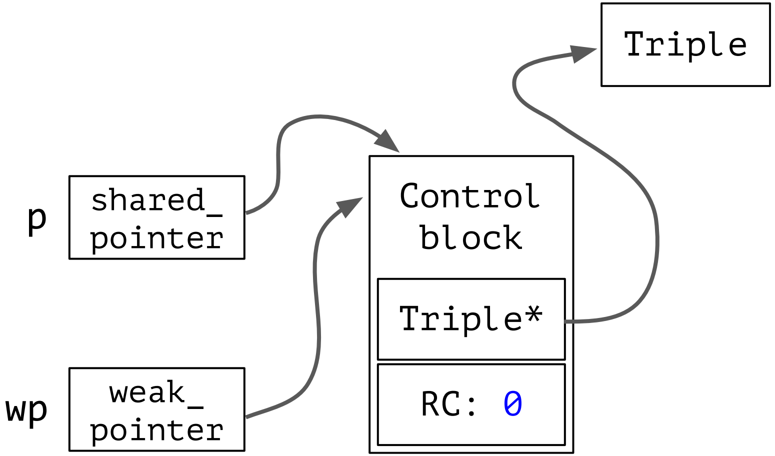
```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    Triple* t = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(t);
    std::weak_ptr<Triple> wp = p;

    print(p);
    p = nullptr;
    return 42;
}
```



When there are also no weak_ptrs to this control block => we can deallocate it as well.

make_shared

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    Triple* t = new Triple{1, 2, 3};
    auto p = std::shared_ptr<Triple>(t);
    std::weak_ptr<Triple> wp = p;

    print(p);
    p = nullptr;
    return 42;
}
```

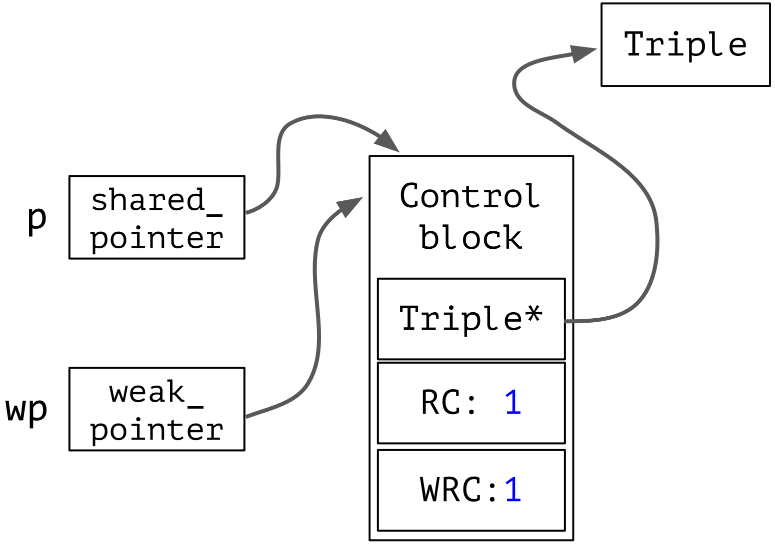


Can't we deallocate object? YES,

Can we deallocate control block? NO,
or incorrect weak_ptr will appear.

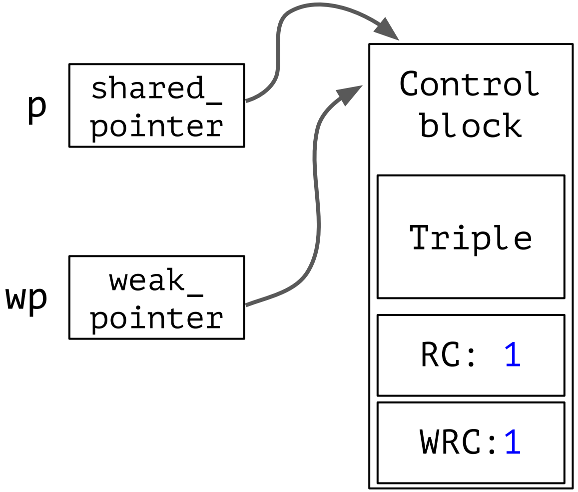
make_shared

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto p =
        std::make_sahred<Triple>(1, 2, 3);
    std::weak_ptr<Triple> wp = p;

    print(p);
    p = nullptr;
    return 42;
}
```



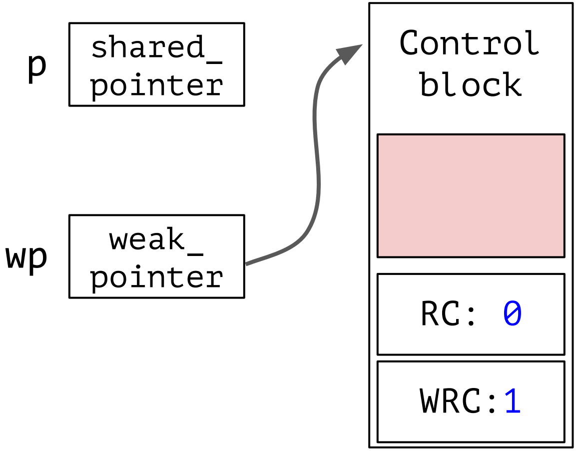
make_shared

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto p =
      std::make_sahred<Triple>(1, 2, 3);
   std::weak_ptr<Triple> wp = p;

   print(p);
   p = nullptr;
   return 42;
}
```



make_shared

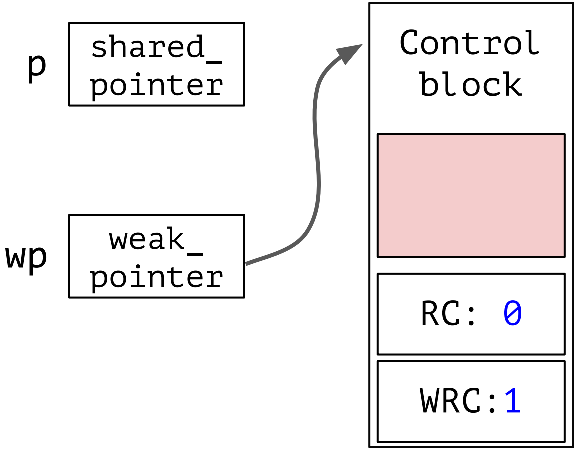
```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto p =
        std::make_sahred<Triple>(1, 2, 3);
    std::weak_ptr<Triple> wp = p;

    print(p);
    p = nullptr;
    return 42;
}
```



make_shared

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
   std::cout << *ptr << std::endl;
}


int foo() {
   auto p =
      std::make_sahred<Triple>(1, 2, 3);
   std::weak_ptr<Triple> wp = p;

   print(p);
   p = nullptr;
   return 42;
}
```

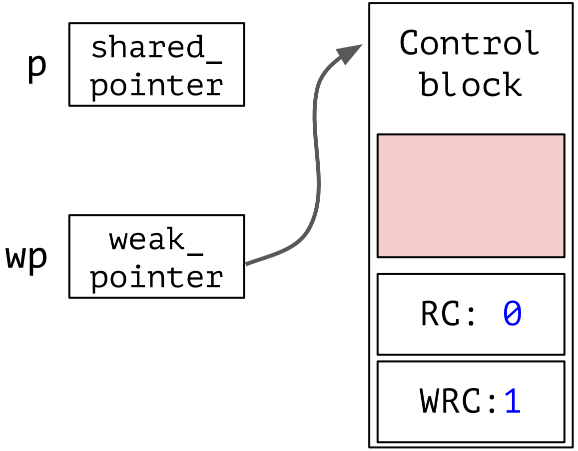
The object itself is dead, but its "dead body" still consumes memory in Control block.



make_shared

```
template <typename T>
void print(std::shared_ptr<T> ptr) {
    std::cout << *ptr << std::endl;
}


int foo() {
    auto p =
        std::make_sahred<Triple>(1, 2, 3);
    std::weak_ptr<Triple> wp = p;

    print(p);
    p = nullptr;
    return 42;
}
```

The object itself is dead, but its "dead body" still consumes memory in Control block.



So, be careful with make_shared it can increase memory drag

Takeaways: smart pointers

Takeaways: smart pointers

- unique_pointers are RAII like helpers to guarde resource (usually memory)

- shared_pointers + weak_pointers are already a system of automatically memory management for C++

Takeaways: smart pointers

- unique_pointers are RAII like helpers to guarde resource (usually memory)

- shared_pointers + weak_pointers are already a system of automatically memory management for C++
  - Not perfect, cycle references can make some troubles, inaccurate creation of shared_ptr, raw pointers still problematic.
  - Still a great alternative for manual memory management.

Takeaways: smart pointers

- unique_pointers are RAII like helpers to guarde resource (usually memory)
- shared_pointers + weak_pointers are already a system of automatically memory management for C++
  - Not perfect, cycle references can make some troubles, inaccurate creation of shared_ptr, raw pointers still problematic.
  - Still a great alternative for manual memory management.
- Both ideas have great potential (ownership in Rust lang, GC in managed languages).

Takeaways: C++ (check that you've learnt)

Takeaways: C++ (check that you've learnt)

- Implementation of basic OOP concepts in C++ (encapsulation, inheritance and polymorphism)

Takeaways: C++ (check that you've learnt)

- Implementation of basic OOP concepts in C++ (encapsulation, inheritance and polymorphism),

- References and value categories,
- Lifetime of objects,
- Copies and move semantics,

Takeaways: C++ (check that you've learnt)

- Implementation of basic OOP concepts in C++ (encapsulation, inheritance and polymorphism),

- References and value categories,
- Lifetime of objects,
- Copies and move semantics,

- Differences between dynamic and static polymorphism, templates and their implementation,

Takeaways: C++ (check that you've learnt)

- Implementation of basic OOP concepts in C++ (encapsulation, inheritance and polymorphism),

- References and value categories,
- Lifetime of objects,
- Copies and move semantics,

- Differences between dynamic and static polymorphism, templates and their implementation,

- Compile time evaluation: metaprogramming and constexpr.

Takeaways: C++ (check that you've learnt)

It is not necessary to become a C++ developer, but learning this language is a great way to understand which problems and challenges can appear in system programming language design.