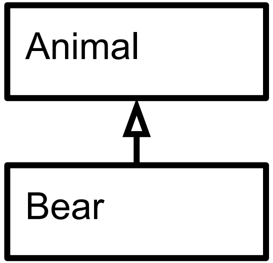
C++ Course 6: Smart Pointers. Exceptions.

By Oleksiy Grechnyev

Polymorphism: Upcasts and Downcasts

class Animal {...};
class Bear: public Animal {...};



Bear inherits (наследует) **Animal**

Upcasts: Every **Bear** is an **Animal** ! Safe and implicit.

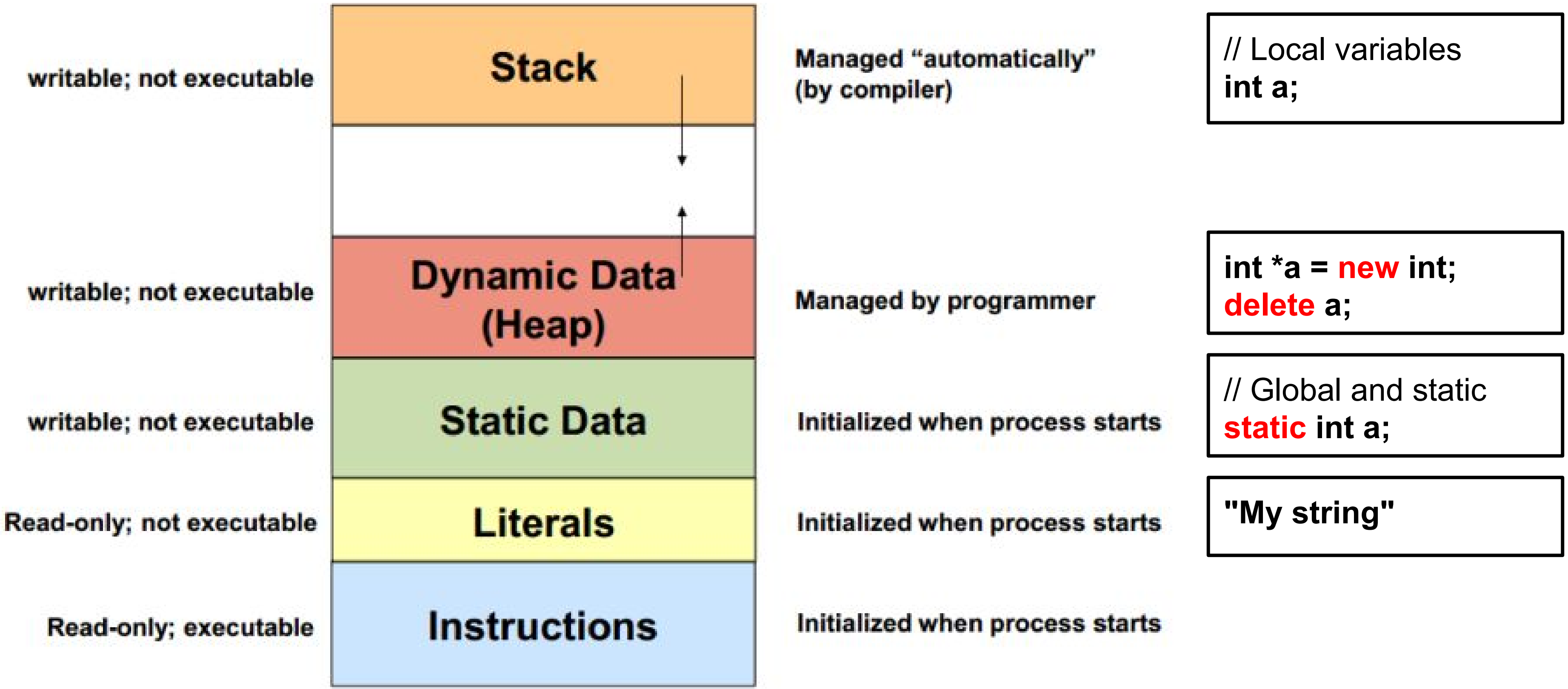
Bear & to **Animal &**, **Bear *** to **Animal ***

Downcasts: Not every **Animal** is a **Bear** !

Animal & to **Bear &**, **Animal *** to **Bear ***

Requires **static_cast** or **dynamic_cast** !
Only works if our **Animal &** or **Animal *** points to a **Bear** object !
Допустимо только если **Animal &** или **Animal *** указывает на объект **Bear** !

C++ memory model



Object Life Cycle

Every C++ object is born and dies
Каждый C++ объект рождается и умирает

Scope	Example	Birth	Death
Local variable	**string s = "Jezebel";**	Definition	The closing }
Temporary object	**string("Jack");**	Code line start	Code line end
Global/static variable	**static string("Jill");**	Program start	Program end
Heap object	**new string("Maria");**	**new**	**delete !!!**
Class field	**string s = "Lilith";**	Class Ctor	Class Dtor
Smart pointers	?	?	?

Working with heap memory: new and delete

Old-style C++ (before 11):

```
void fun(){
	string * pS = new string("Some text");
	..
	delete pS;   // Don't forget delete !!! Otherwise a memory leak !
}

class A{
public:
	A(const char * s) : pS(new std::string(s)) {}   // Ctor
	...
	~A(){ delete pS;}   // Dtor
private:
	string * pS;
};
```

Trouble with heap objects 1

```
string * pS = new string("Some text");
```

If we forget delete: Memory Leak!

If we put delete twice : double delete. Program crashes!

```
delete pS;
delete pS;
```

Things that make it worse : multiple returns, exceptions.

```
void fun(){
	string * pS = new string("Some text");
	..
	if (...) return;  // Forgot delete here ! Memory leak !!!
	...
	if (..) throw runtime_error("HAHA !");   // Forgot delete here ! Memory leak !!!
	delete pS;
}
```

Trouble with heap objects 2

There is no way to tell if a pointer points to a valid heap object:
Невозможно проверить указывает ли указатель на активный heap объект:

```
string s1("Nel Zelpher");
string *pS1 = &s1;
delete s1; // Wrong ! Not a heap object!
string *pS2 = new string("Sophia Esteed");   // Created a heap object
delete pS2; // Deleted it. OK !
delete pS2; // Error ! Double delete !
int *pI = new int;  // int heap object
string *pS3 = (string *) pI;  // pS3 points to an int heap object (Wrong type !)
delete pS3; // Error ! Wrong type !
```

My program crashes, why ??????
No way to check for these situations !!!

Solution: unique_ptr

unique_ptr takes a heap object under *exclusive ownership* (исключительное владение)

```
unique_ptr<string> u = make_unique<string>("Abracadabra");
```

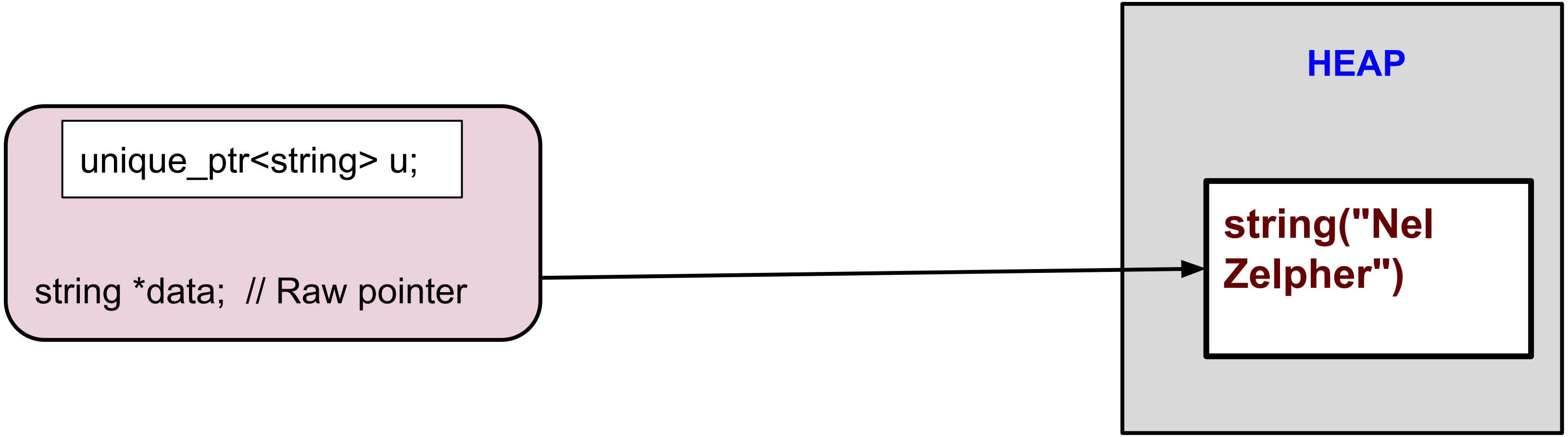
or

```
unique_ptr<string> u(new string("Abracadabra"));
```

It a thin and efficient wrapper around a raw pointer. Roughly speaking:

```
template <typename T>
class unique_ptr{
public:
        unique_ptr(const T & data): data(data) {}  // Ctor
        ~unique_ptr() {delete data;}  // Managed object is deleted when unique_ptr dies !
        ...
private:
        T * data;       // Pointer to the managed object
};
```

unique_ptr manages a heap object



unique_ptr manages a heap object (управляет хип - объектом)
While **unique_ptr** lives, the managed object lives
Пока живет **unique_ptr** живет и хип-объект
When **unique_ptr** dies, its destructor deletes the managed object
Когда **unique_ptr** умирает, его деструктор удаляет управляемый объект
unique_ptr has the size of a *raw pointer* (typically 8 bytes)

Using heap with unique_ptr

```
void fun(){
        unique_ptr<string> u = make_unique<string>("Some text");
        ..
        // The string object is deleted here by the unique_ptr destructor!
}    // The closing brace: u runs out of scope here

class A{
public:
        A(const char * s) : uS(new std::string(s)) {}   // Ctor
        ...
        // No Destructor. The managed object is deleted automatically !
private:
        unique_ptr<string> uS;
};
// No more delete operators anywhere ! Managed objects die together with uniqie_ptr !
```

What not to do !

```
string *pS = new string("Maria Traydor");  // Created a heap object  with new !
unique_ptr<string> uS1(pS);  // Created a unique_ptr out of it
```

Now **uS** has exclusive ownership of the heap object !
Теперь **uS** исключительно владеет этим объектом в хипе!

DON'T DO THIS:

```
delete pS;   // Wrong ! unique_ptr takes care of it ! Double delete !
unique_ptr<string> uS2(pS);  // Wrong ! Creating uS2 for the same object !!!
```

Cannot create 2 **unique_ptr** objects from a single heap object !!! Double delete !
make_unique makes it safer:

```
auto uS1 = make_unique<string>("Maria Traydor");
```

Using unique_ptr 1

```
auto u = make_unique<string>("Maria Traydor");
```

What can we do with it ? Use it as a normal pointer!
Что с ним делать? Использовать как обычный указатель!
Operators **`*u`, `u->`** are defined. **`*u`** is the underlying **string** object.

```
cout << "*u = " << *u << endl;  // Dereferencing
*u = "Nel Zelpher";  // Change the string (NOT pointer !)
cout << "*u = " << *u << endl;  // Print the string again !
cout << " u->size() = " << u->size() << endl;  // Call method
cout << " (*u).size() = " << (*u).size() << endl;  // The same !
```

Check that **unique_ptr** object has a managed object (not **nullptr**):

```
if (u)
    ...
```

Using unique_ptr 2

```
auto u = make_unique<string>("Maria Traydor");

u.reset();                      // Force delete
u = nullptr;                    // The same
string *p1 = u.get();           // Get the raw pointer (Don't delete it !)
delete p1; // Error !!!
string *p2 = u.release();       // Release ownership of the managed object
delete p2;                      // Now you must delete it !
```

unique_ptr cannot be copied but can be moved !

```
auto u2 = u;                 // Error !!!
auto u2 = move(u);           // OK ! Ownership transferred to u2!
```

move transfers ownership of the managed object to another **unique_ptr** object.
Операция **move** передает владение хип-объектом другому объекту **unique_ptr**.
The old object **u** is reset (set to **nullptr**).

unique_ptr and polymorphism (upcasts, downcasts) ?

Bear is a subclass of **Animal**. Can we do something like this ???

```
auto upB = make_unique<Bear>("Teddy", 7);
auto upA = dynamic_cast<unique_ptr<Animal> >(upB);
```

unique_ptr and polymorphism (upcasts, downcasts) ?

Bear is a subclass of **Animal**. Can we do something like this ???

```
auto upB = make_unique<Bear>("Teddy");
auto upA = dynamic_cast<unique_ptr<Animal> > (upB);   // Wrong !!!
```

NO !!! This would be like copying **unique_ptr** !

The correct way: use ***upB** as a reference or **upB.get()** as a raw pointer:

```
Animal & rA = *upB;             // Reference upcast. Good !
Animal * pA = upB.get();        // Pointer upcast. Ugly ! Don't delete pA !
```

unique_ptr and source (factory) functions

Source (factory) creates a heap object and returns **unique_ptr**

Источник (фабрика) создает хип-объект и возвращает **unique_ptr**

```
unique_ptr<Tjej> factory1(const string & name){
    return make_unique<Tjej>(name);  // No need for explicit move here
}
```

```
unique_ptr<Tjej> factory2(const string & name){
    unique_ptr<Tjej> upT = make_unique<Tjej>(name);
    return move(upT);   // Will work without move also
}
```

Usage:

```
    auto upT1 = factory1("Nel Zelpher");
    auto upT2 = factory2("Claire Lasbard");
    ...
}   // upT1, upT2 and managed objects are destroyed HERE !
```

unique_ptr and sinks (consumers)

Sink (consumer) eats **unique_ptr** and destroys it (and the managed object)

Потребитель пожирает **unique_ptr** и убивает его (и управляемый объект)

Argument is passed *by value* (NOT reference !) using **move** :

```
void sink(unique_ptr<Tjej> upT){     // By value !!! NO ref ! MOVED here !
     // upT is local in this function
     cout << "Sink: name = " << upT->getName() << endl;
     // Now upT and the managed object are destroyed as they go out of scope !
}
```

Usage:

```
     auto upT = make_unique<Tjej>("Sophia Esteed");
     sink(move(upT));                                    // Move to sink

     // The managed object is destroyed by now and upT == nullptr
     ...
}   // Nothing happens here !
```

Smart pointer which can be copied ?

ONE **unique_ptr** manages *ONE* heap object :

unique_ptr<string> **u**;

HEAP

string("Nel Zelpher")

MANY copies together manage *ONE* heap object :

p1

p2

p3

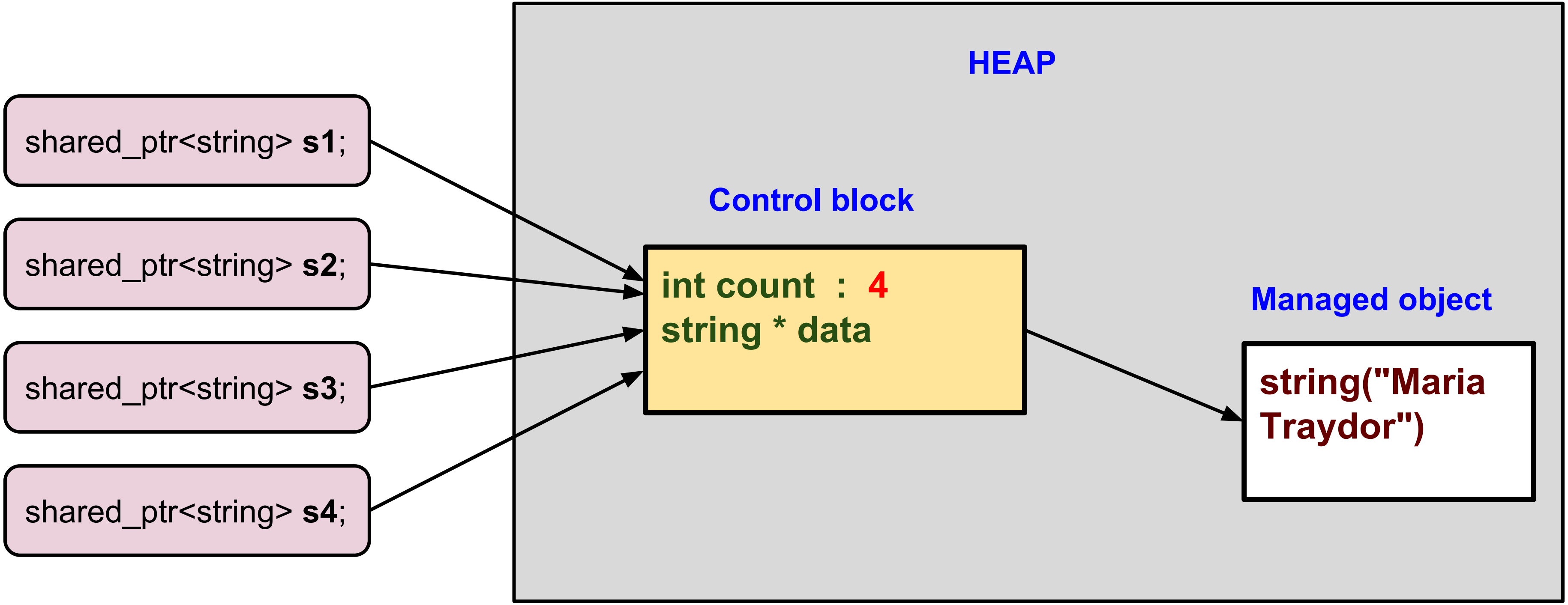
string("Maria Traydor")

While *at least one* copy is alive, the managed object lives.

When *the last* copy dies, the managed object is deleted.

Хип-объект умирает вместе с *последней* копией.

shared_ptr concept



While *at least one* copy is alive, the managed object AND the control block live.
When *the last* copy dies, they BOTH are deleted.

Creating shared_ptr

Creating **shared_ptr** :

```
shared_ptr<Tjej> s1 = make_shared<Tjej>("Maria Traydor");
auto s2 = make_shared<Tjej>("Nel Zelpher");
shared_ptr<Tjej> s3(new Tjej("Sophia Esteed"));
```

You can also convert **unique_ptr** to **shared_ptr** (don't forget **move** !)

```
auto u = make_unique<Tjej>("Mirage Koas");
shared_ptr<Tjej> s4(move(u));    // u is destroyed, s4 takes over the object
```

shared_ptr can be both copied and moved:

```
auto s5 = s1;
auto s7 = move(s3);
s3 = s2;  // Copy pointer, not value

*s1 = *s3;  // Copy value, not pointer!
```

Using shared_ptr

Use it as a normal pointer!

```
auto s = make_shared<string>("Kajsa");
cout << "*s = " << *s << endl;
*s = "Eva";   // Change the value, not ptr !
cout << "*s = " << *s << endl;
cout << "s->size() = " << s->size() << endl;
```

Other things you can do:

```
s.reset();   // Resets s, does not delete the managed object unless it's the last copy
s = nullptr;  // The same
cout << s.use_count() << endl;   // Number of copies in existence
cout << s.unique() << endl;        // Is this the only copy ?
string *str = s.get();                     // Get a raw pointer
if (s)                                     // Check that s is valid (not nullptr)
    ...
```

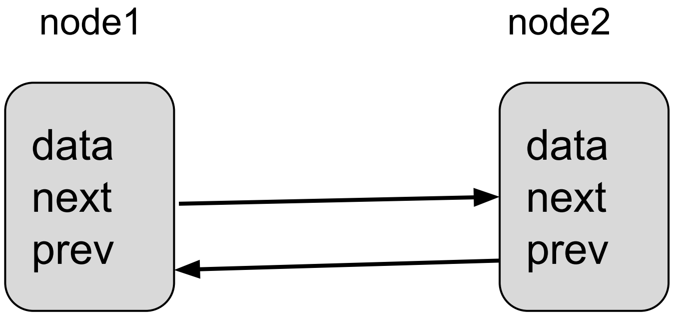
shared_ptr and polymorphism

```
auto sB = make_shared< Bear>("Teddy");   // Create a shared_ptr<Bear> object

cout << "Ref upcast :" << endl;
Animal & rA = *sB;
rA.talk();

cout << "Raw ptr upcast :" << endl;
Animal * pA = sB.get();                          // Get the raw Bear * ptr
pA->talk();

cout << "shared_ptr upcast :" << endl;
shared_ptr<Animal> sA = sB;                        // Implicit upcast
sA->talk();

cout << "shared_ptr downcast :" << endl;
shared_ptr<Bear> sB1 =  static_pointer_cast<Bear> (sA);     // No checks !
shared_ptr<Bear> sB2 =  dynamic_pointer_cast<Bear> (sA);    // Checks. Safe.
sB1->talk();
sB2->talk();
```

Can shared_ptr create a memory leak ?

Suppose we want a double-linked list of nodes:

```
struct Node{
    Node(const string & data) : data(data) {}
    string data;
    shared_ptr<Node> next;      // Next node
    shared_ptr<Node> prev;      // Previous node
};
```

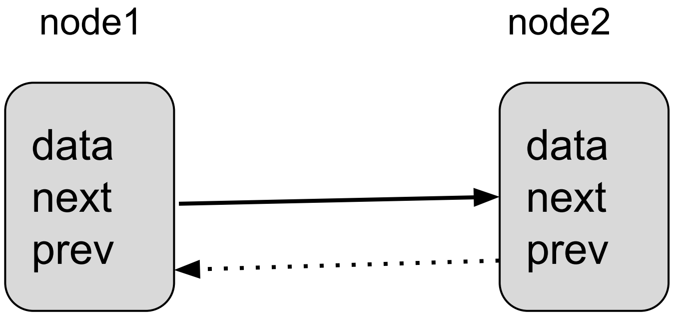


shared_ptr cyclic reference ! Memory leak !!!

Solution: weak_ptr

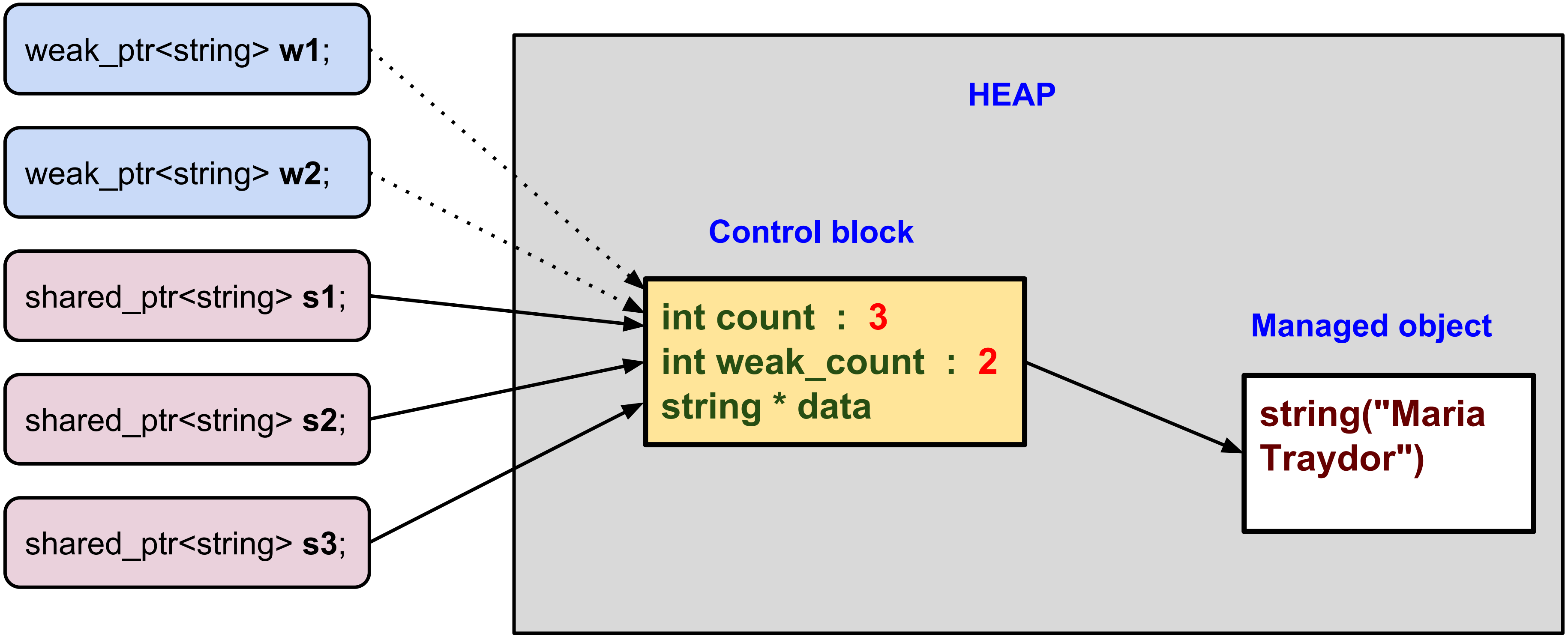
Let us rewrite our node like this:

```
struct Node{
    Node(const string & data) : data(data) {}
    string data;
    shared_ptr<Node> next;      // Next node
    weak_ptr<Node> prev;        // Previous node : weak_ptr !!!!
};
```



No more **shared_ptr** cyclic reference ! No more memory leak !!!

weak_ptr concept: a weak reference to a shared_ptr



When the last **shared_ptr** dies, the managed object is deleted, the control block stays !

When the last **weak_ptr** dies, the control block is deleted !

Using weak_ptr

Create a **weak_ptr** out of a **shared_ptr** :

```
shared_ptr<string> s = make_shared<string>("Maria Traydor");
weak_ptr<string> w = s;
```

Restore **shared_ptr** from a **weak_ptr** (creates one more **shared_ptr** copy):

```
if (! w.expired() ) {                                   // Check if w is expired
        shared_ptr<string> s = w.lock();        // Returns nullptr if expired
        ...
}
```

weak_ptr is *expired* when the last **shared_ptr** dies and the managed object is deleted

weak_ptr can be reset, copied and moved, but cannot be dereferenced:

```
*w              // Error !!!
w->size()       // Error !!!
```

weak_ptr can be also used for caches, observers (like **WeakReference** in Java)

Exceptions

Exception interrupts the flow of the program

Исключение прерывает поток выполнения программы

```
if (a >= 0) throw std::runtime_error("The user is an idiot !");
```

Exception can be caught

Исключение может быть поймано

```
try {
        ...
} catch (const std::exception & e) {
        cerr << e.what() << endl;
}
```

In the exception is NOT caught, the program terminates

Если исключение не поймано, программа завершается

Objects of any type can be thrown and caught

```
try {
        throw 17;                               // int
        throw (unsigned)17;          // unsigned, not int !
} catch (int i) {
        cout << "int exception " << i << endl;
} catch (double d) {
        cout << "double exception " << d << endl;
} catch (const string & s) {
        cout << "string exception " << s << endl;
} catch (const char * cS) {          // Will catch string literals
        cout << "char * exception " << cS << endl;
} catch (...) {                    // Default
        cout << "Unknown exception" << endl;
}
```

NEVER EVER use **new** in **throw** !!!! Memory leak !!!

But the standard class is std::exception and subclasses

throw runtime_error("Earthquake"); // std::string parameter !
throw logic_error("Earthquake");
// invalid_argument, domain_error, length_error, out_of_range
// range_error, overflow_error, underflow_error

Define your own exception: Inherit **runtime_error** (simplest), or implement **std::exception**
what() message is printed if not caught (**std::exception** subclasses only !)

```
struct DiamondException : public std::runtime_error{
    explicit DiamondException(const std::string & s) :
         runtime_error("Diamond: " + s) {}
};

struct SapphireException : public std::exception{
    const char * what() const noexcept override {
         return "Al2O3";
    }
};
```

Exceptions and function calls

```
void f3() {
	int a;
	throw runtime_error("HAHA");
}
void f2() {
	string s("Local String"); f3();
}
void f1(){ f2(); }
void main() {
	try {
		f1();   // Calls f1() -> f2() -> f(3)
	} catch (...) { /* Some code*/   }
}
```

Exception thrown in **f3** goes right through **f2** and **f1** and is caught in **main**.

Any local variables of **f3**, **f2**, **f1** are properly deleted (*stack unwinding*)

Re-throw and noexcept

Re-throw the exception just caught:

```
catch (...) {
	cout << "Caught something !!!" << endl;
	throw ;
}
```

noexcept : Specifies that a function/method does not throw.
It includes the exceptions *passing through* (f1() -> f2() -> f(3)).

```
int add(int a, int b) noexcept {
	return a+b;
}
```

noexcept allows for better optimization. If the function throws anyway, it cannot be caught and the program terminates.

Thank you for your attention !

title

text