System Software Crash Couse

Samsung Research Russia
Moscow 2019

Block G: Advanced C++

11. Smart Pointers

Eugene Zouev

Smart pointers

Before we start…

Breaking News:

Pointers are to be removed from the C++2023!!!

«Комитет по стандартизации языка в Джексонвиле две недели назад принял решение о том, что **указатели будут объявлены устаревшими** в C++20 и **с большой долей вероятности будут удалены** из C++23.»

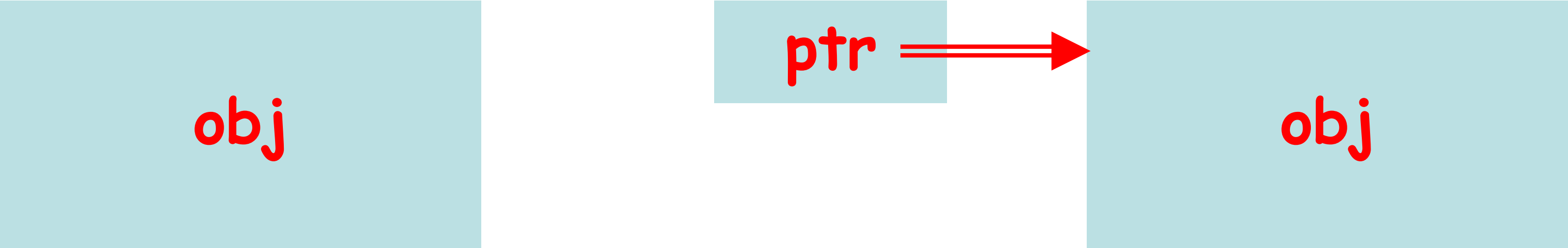
1 апреля

https://habrahabr.ru/post/352570/

Problems with usual C++ pointers

```
T obj;
```

```
T* ptr;
```



obj

ptr

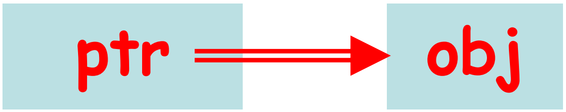
obj

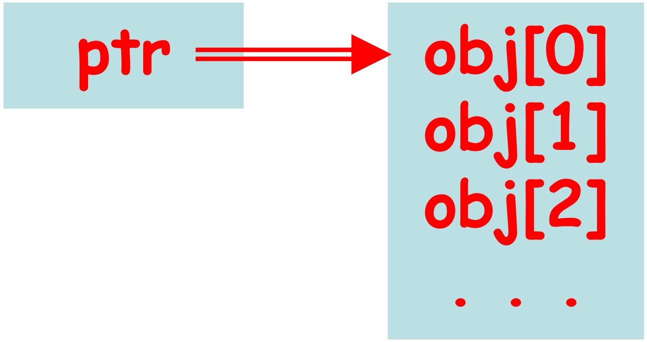
The problems with pointers come from its low-level nature…

Problems with usual C++ pointers

Scott Meyer:
6 kinds of problems with pointers

Problems 1 & 4:
A pointer can point either to a **single object**, or to an **array**.





```
delete ptr;
```

```
delete[] ptr;
```

Problems with usual C++ pointers

Problem 2:
A declaration of a pointer tells nothing whether we must destroy the object pointed after the work is completed.

Or: does the pointer **owns the object** pointed?

```
void fun(T* ptr)
{
  // Some work with an object
  // pointed to by ptr.

  // Should we destroy the object
  // before return?
  return;
}
```

Problems with usual C++ pointers

Problem 3:
Even if we know that we should destroy the object pointed to by a pointer - in general we don't know **how to do that**!

I.e., either just to apply **`delete`** or use some special function for that?

```
void fun(T* ptr)
{
   // Some work with an object
   // pointed to by ptr.

   // We know that fun should destroy
   // the object before return.
   delete ptr;
   return;
}
```

…or perhaps:

```
myLib::myDelete(ptr)
```

Problems with usual C++ pointers

Problem 5 (a consequence from problem 2): Even if we **own** the object pointed to by a pointer it's hard (or even impossible) provide **exactly one** act of destroy.

I.e., it's quite easy either to leave the object live, or to try to destroy it twice or more.

```
void lib_fun(T* ptr)
{
   // This library performs some
   // actions on the object passed
   // as parameter.

   // The function doesn’t destroy
   // the object before return.
   return;
}
```

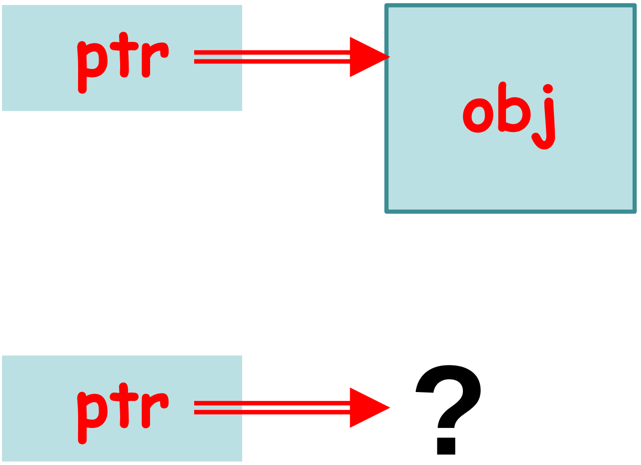
```
void user_fun()
{
   T* ptr = new T();
   // The function owns its object.

   lib_fun(ptr);
   // Should we destroy the object
   // before return, OR lib_fun has
   // already destroyed it??
   return;
}
```

Problems with usual C++ pointers

Problem 6:
There is no way to check whether a pointer actually points to a real object.

Or: to check whether the pointer is “dangling pointer“.

```
T* ptr = new T();
...
if ( condition ) delete ptr;
...
// Long code…
...
// How to know whether ptr
// still points to an object?
...
```



Problems with usual C++ pointers

Problem 7 (in addition to Scott Meyers' ☺):
There is no way to ensure that an object gets destroyed when the single pointer to it disappears.

```
if ( condition )
{
   T* ptr = new T();
   ...
   // No delete ptr
}
...
```

Here, `ptr` doesn't exist, but the object still does: **memory leak**

Solution: Smart pointers

Since C++11

Four standard templates

C++98

auto_ptr

unique_ptr

shared_ptr

weak_ptr

Obsolete, outdated;
was transformed
to unique_ptr

From now on, there is a kind of “mauveton” (bad taste) to use *-like pointers ☺.

Do you remember **casts**?
The common C-like construct **(T)expr**
was “splitted” into four specific kinds:
static_cast, dynamic_cast, const_cast,
and reinterpret_cast…

Solution: Smart pointers

Since C++11

Four standard templates

`unique_ptr`

`shared_ptr`

`weak_ptr`

- All templates are **wrappers** over classic C++ pointers.
- All templates preserve major pointer's functionality (at least dereferencing).
- Each `xxx_ptr` template adds some extra ("smart") functionality.
- Each template is implemented without loss of efficiency (almost ☺) comparatively with classical pointers.

Smart pointers: a General Idea

```
// A simple smart pointer template
template <typename T>
class smart_pointer
{
  T* obj;  // The “raw” C++ pointer
public:
  // Constructor accepts the object
  // that will be “owned” by smart_pointer
  smart_pointer(T* o) : obj(o) { }
  // Destructor guarantees that the object
  // will be destroyed when leaving the scope
  // of smart_pointer object
  ~smart_pointer() { delete obj; }

  // Overloaded -> selector
  T* operator->() { return obj; }

  // Overloaded dereferencing operator
  T& operator* () { return *obj; }
}
```

The task for your homework:
- Add some operators that make `smart_pointer` look more similar to usual C++ pointers.
- Write an example that shows the advantage of such a template.

RAII pattern:
“Resource
Acquisition
Is
Initialization”

Smart pointers: unique_ptr

Replaces auto_ptr

unique_ptr implements the semantics of **exceptional ownership:**
At any execution point, **only one pointer "owns" the object pointed**

```
std::unique_ptr<int> x(new int(42));
std::unique_ptr<int> y;

y = x;

std:unique_ptr<int> z(x);
```

Compile-time error

Compile-time error

Question for your home work:
- How it is implemented?
(No details, just give an idea of the implementation)

Smart pointers: unique_ptr

Sometimes it's necessary to "pass" ownership rights to some other pointer:

```
std::unique_ptr<int> x(new int(42));
std::unique_ptr<int> y;

y = x;
```

```
y = std::move(x);
```

Transfers ownership rights to the y pointer **and nullifies** x

Question:
- Is it possible to create and delete **arrays** using unique_ptr? If yes, how?

Some extra functionality:

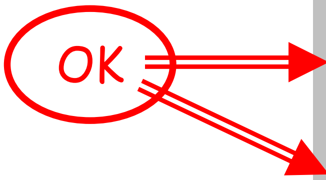
`x.reset()` nullifies ownership rights
`x.get()` returns "raw" C++ pointer

Smart pointers: shared_ptr

shared_ptr implements the semantics of **cooperative ownership**: several pointers can share the same object.

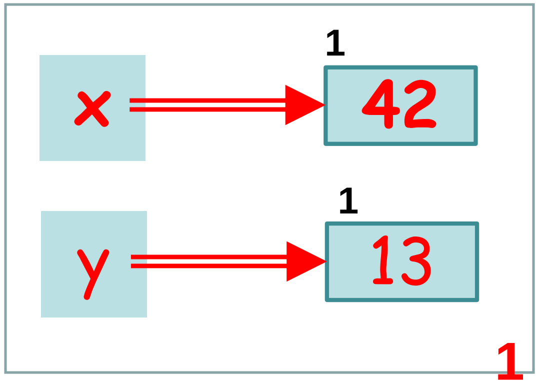
The object is automatically destroyed when there is no (more) pointers to it.

This is the **ARC mechanism**: **automatic reference counting**. So, it can be treated as a kind of garbage collector implementation.

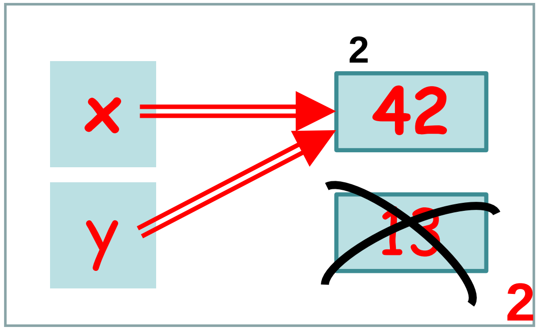


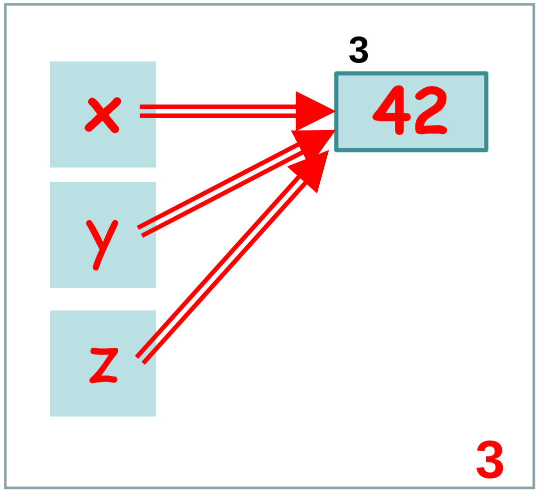
```
std::shared_ptr<int> x(new int(42));
std::shared_ptr<int> y(new int(13));

y = x;

std:shared_ptr<int> z(x);
```

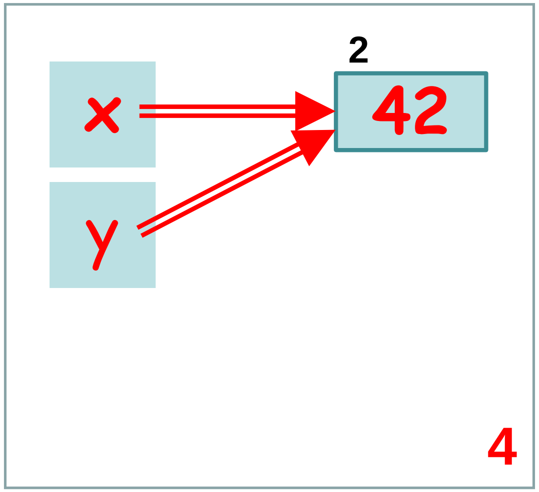
shared_ptr: An Example



```
  std::shared_ptr<int> x(new int(42));
  std::shared_ptr<int> y(new int(13));
1 ⇒
  y = x;
2 ⇒
  {
    std:shared_ptr<int> z(x);
3 ⇒
  }
4 ⇒
```







Question for your homework:
Where reference counter is stored, and why? Options:
- Together with the pointer
- Together with the object pointed
- Somewehe in dynamic memory

Smart pointers: shared_ptr

Usual functionality is provided: reset(), get() functions as for unique_ptr, and additionally:

- **operator bool()** to check if a pointer is valid
- Complementary function std::make_shared<T>()

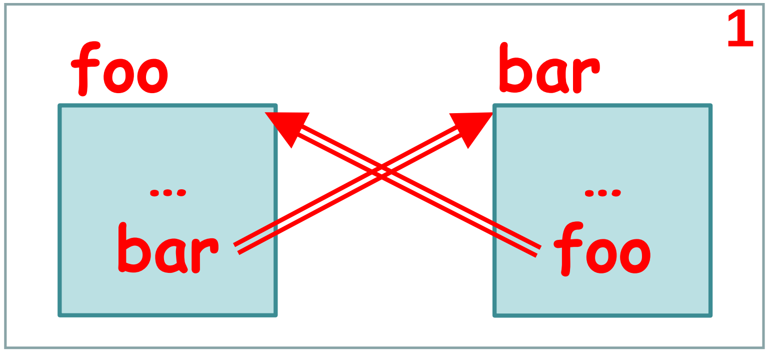
Problems with shared_ptr:

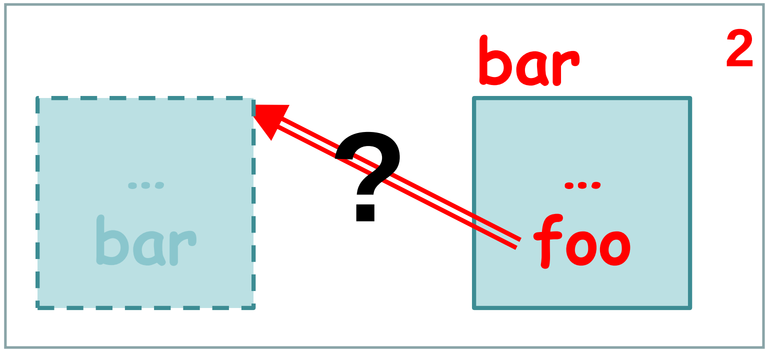
- An overhead: shared_ptr is represented by **two pointers** (try to explain why)
- A lot of wording: awkward notation (use **auto**)
- Some cases with undefined behavior & exceptions (use make_shared)
- Circular references!

Smart pointers: shared_ptr

Circular references with **raw pointers**

```
class Bar;

class Foo {
 public:
   Foo() { ... }
  ~Foo() { ... }
   Bar* bar;
};

class Bar {
 public:
   Bar() { ... }
  ~Bar() { ... }
   Foo* foo;
};
...
void fun() {
  auto foo = new Foo();
  foo->bar = new Bar();
  foo->bar->foo = foo;    // 1
  delete foo;             // 2
}
```

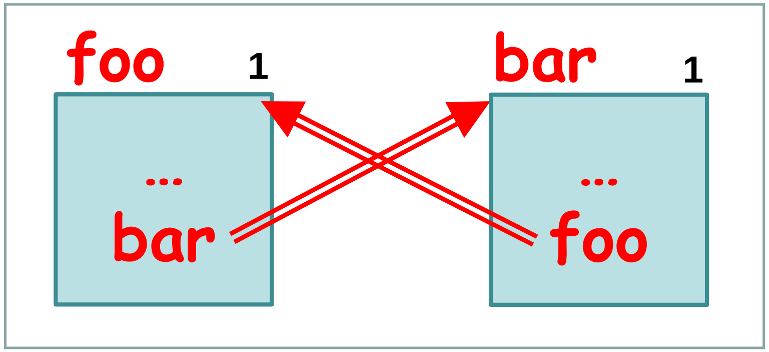




Smart pointers: shared_ptr

Circular references with shared_ptr

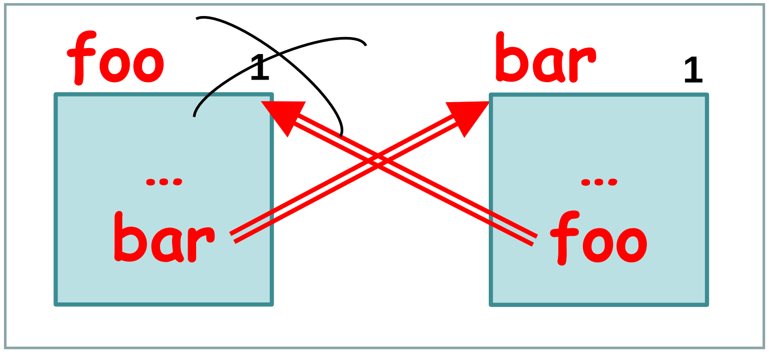
```
class Bar;

class Foo {
 public:
   Foo() { ... }
  ~Foo() { ... }
   std::shared_ptr<Bar> bar;
};

class Bar {
 public:
   Bar() { ... }
  ~Bar() { ... }
   std::shared_ptr<Foo> foo;
};
...
void fun() {
  auto foo = std::make_shared<Foo>();
  foo->bar = std::make_shared<Bar>();
  foo->bar->foo = foo;
  delete foo;  // No result
}
```



Smart pointers: weak_ptr

Circular references with weak_ptr

```
class Bar;

class Foo {
 public:
   Foo() { ... }
  ~Foo() { ... }
   std::shared_ptr<Bar> bar;
};

class Bar {
 public:
   Bar() { ... }
  ~Bar() { ... }
   std::weak_ptr<Foo> foo;
};
...
void fun() {
  auto foo = std::make_shared<Foo>();
  foo->bar = std::make_weak<Bar>();
  foo->bar->foo = foo;
  delete foo;  // OK!!
}
```



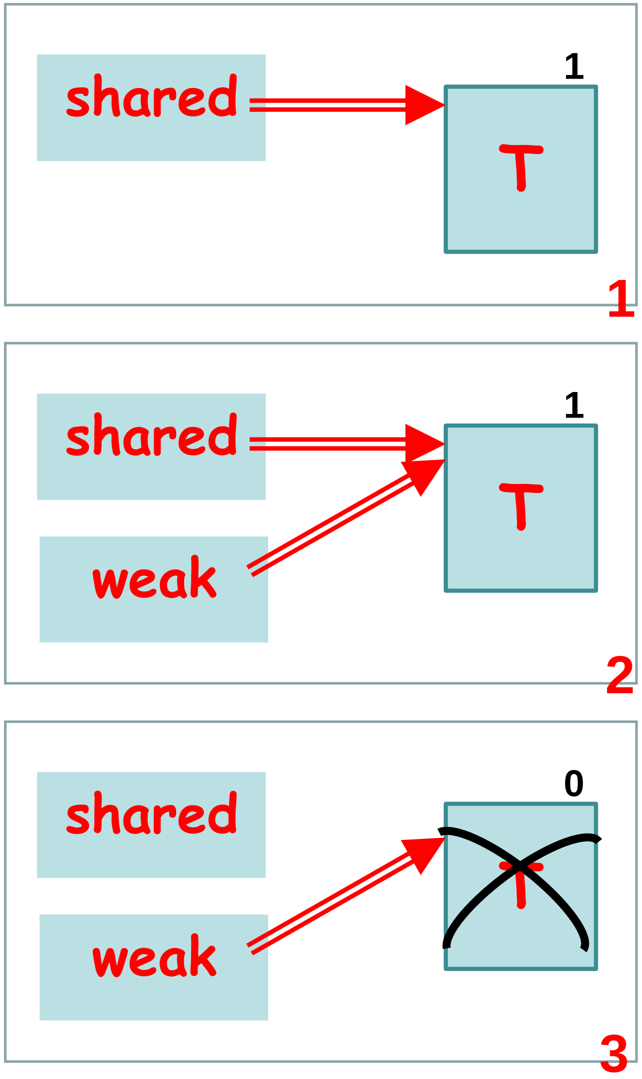
Smart pointers: weak_ptr

A complementary notion to unique_ptr:

- No dereferencing operator
- No check for "null"

```
// Suppose T is some type

auto shared = std::make_shared<T>();//1
...
std::weak_ptr<T> weak(shared); // 2
...
shared = nullptr;      // 3
...
if ( weak.expired() ) ...
```

Question for your homework: Try to explain why weak_ptr is necessary?



Smart pointers: references

C++ Standard, Sect. 23.11 (smart pointers).

Scott Meyers, Effective Modern C++, Chapter 4.

http://archive.kalnytskyi.com/2011/11/02/smart-pointers-in-cpp11/ (Russian)

https://habrahabr.ru/post/140222/ (Russian; incomplete)

http://umich.edu/~eecs381/handouts/C++11_smart_ptrs.pdf *Very informative paper with examples and pictures*

https://mbevin.wordpress.com/2012/11/18/smart-pointers/