Программирование на С++. Часть 2
Лекция 13

ПМИ Семестр 2

Демяненко Я.М.

2026

Жалуемся на работу с указателями

Основные проблемы

Утечкой памяти называется ситуация, когда в программе нет ни одного указателя, хранящего адрес объекта, созданного в динамической памяти.

Висячей ссылкой называется указатель, ссылающийся на уже удалённый объект.

Умные указатели (начиная с C++11)

Smart pointer — это объект, работать с которым можно **как с обычным** указателем, но в отличие от последнего, он предоставляет некоторый **дополнительный функционал** (например, автоматическое **освобождение** закрепленной за указателем области **памяти**).

или

Smart pointer — класс (обычно шаблонный), имитирующий интерфейс обычного указателя и добавляющий некую новую функциональность, например проверку границ при доступе или очистку памяти.

Предназначение

Умные указатели призваны для борьбы с утечками памяти, которые сложно избежать в больших проектах.

Они особенно удобны в местах, где возникают исключения, так как при последних происходит процесс раскрутки стека и уничтожаются локальные объекты.

В случае обычного указателя — уничтожится переменная-указатель, при этом ресурс останется не освобожденным.

В случае умного указателя — вызовется деструктор, который и освободит выделенный ресурс.

Семантика владения

Чаще всего умный указатель **инкапсулирует семантику владения ресурсом**.
В таком случае он называется **владеющим** указателем.

Владеющие указатели применяются для борьбы с утечками памяти и висячими ссылками.

Семантика владения для динамически созданных объектов означает, что удаление или присвоение нового значения указателю будет **согласовано с временем жизни** объекта.

Умные указатели

- unique_ptr
- shared_ptr
- weak_ptr

Все они объявлены в заголовочном файле <memory>

unique_ptr — замена auto_ptr

Этот указатель пришел на смену старому и проблематичному **auto_ptr**.

Основная проблема **auto_ptr** заключается **в правах владения**.

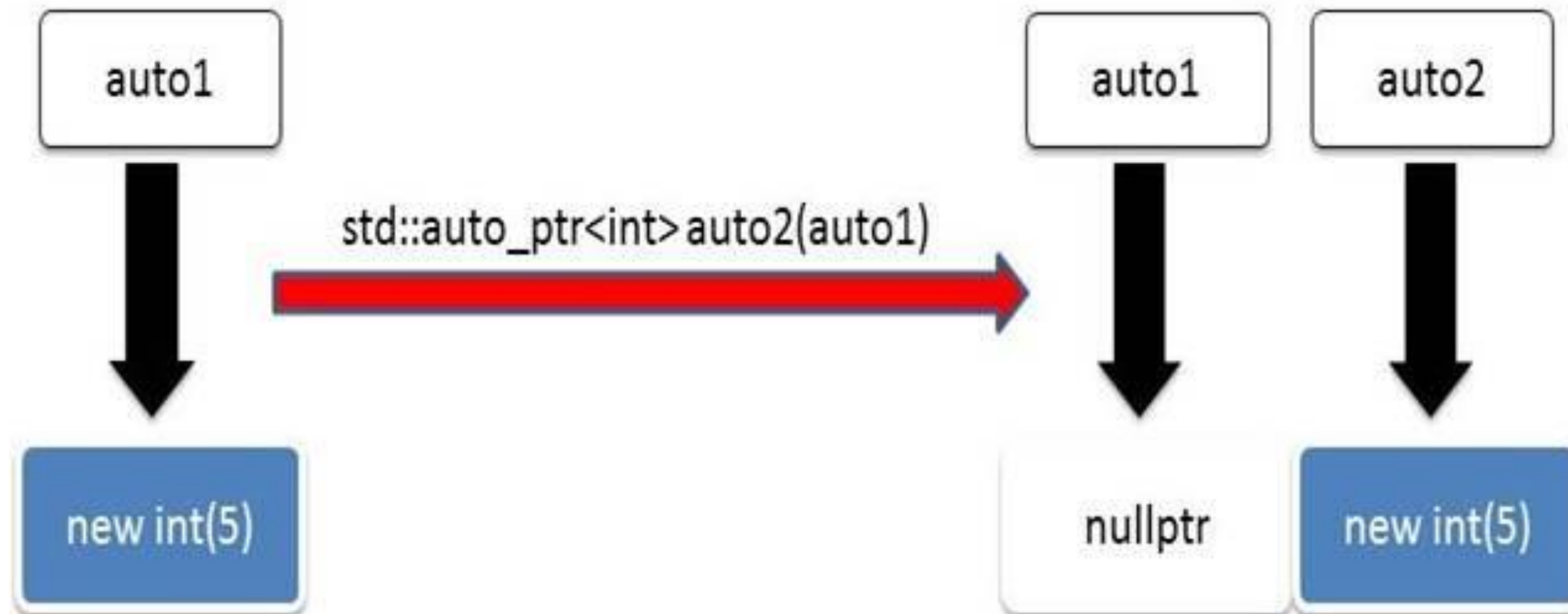
Объект этого класса теряет права владения ресурсом при копировании (присваивании, использовании в конструкторе копий, передаче в функцию по значению).

auto_ptr (deprecated)

- Реализует стратегию «одного владельца»
- При копировании владелец передается

Что означает?

```
Picture* a = new Picture();
Picture* b = a;

std::auto_ptr<int> x_ptr(new int(5));
std::auto_ptr<int> y_ptr;

// вот этот нехороший и неявный момент

y_ptr = x_ptr;

// права владения ресурсов уходят в y_ptr и x_ptr начинает указывать на null pointer

//после этого x_ptr перестает указывать на объект

std::cout << *x_ptr << std::endl;      // error
```


auto1
std::auto_ptr<int>auto2(auto1)
new int(5)
auto1
auto2
nullptr
new int(5)

Неудобства при работе с контейнером

Это очень неудобно, при работе с контейнером из умных указателей.

Банальное

```
std::vector<std::auto_ptr<int> > vec;
// ...
std::auto_ptr<int> tmp = vec[0];
```

сделает элемент вектора невалидным.

unique_ptr

В отличит от auto_ptr, unique_ptr **запрещает копирование**

```
std::unique_ptr<int> x_ptr(new int(42));
std::unique_ptr<int> y_ptr;

// ошибка компиляции
y_ptr = x_ptr;

// ошибка компиляции
std::unique_ptr<int> z_ptr(x_ptr);
```

Изменение прав владения

Изменение прав владения ресурсом осуществляется с помощью вспомогательной функции std::move (которая является частью механизма перемещения).

```
std::unique_ptr<int> x_ptr(new int(42));
std::unique_ptr<int> y_ptr;

// также, как и в случае с  auto_ptr, права владения переходят к y_ptr, а x_ptr начинает указывать на
// null pointer
y_ptr = std::move(x_ptr);
```

Методы unique_ptr

Как auto_ptr, так и unique_ptr обладают методами:

- reset(), который сбрасывает права владения,
- get(), который возвращает сырой (классический) указатель.

```
std::unique_ptr<Foo> ptr = std::unique_ptr<Foo>(new Foo);

// получаем классический указатель
Foo *foo = ptr.get();
foo->bar();

// сбрасываем права владения
ptr.reset();
```

unique_ptr против auto_ptr

unique_ptr недалеко ушел от своего предшественника в плане удобства использования,

но, во всяком случае, он обезопасил от неявных смен прав владений ресурсом.

shared_ptr

Это самый популярный и самый широко используемый умный указатель.

Он начал своё развитие как часть библиотеки boost.

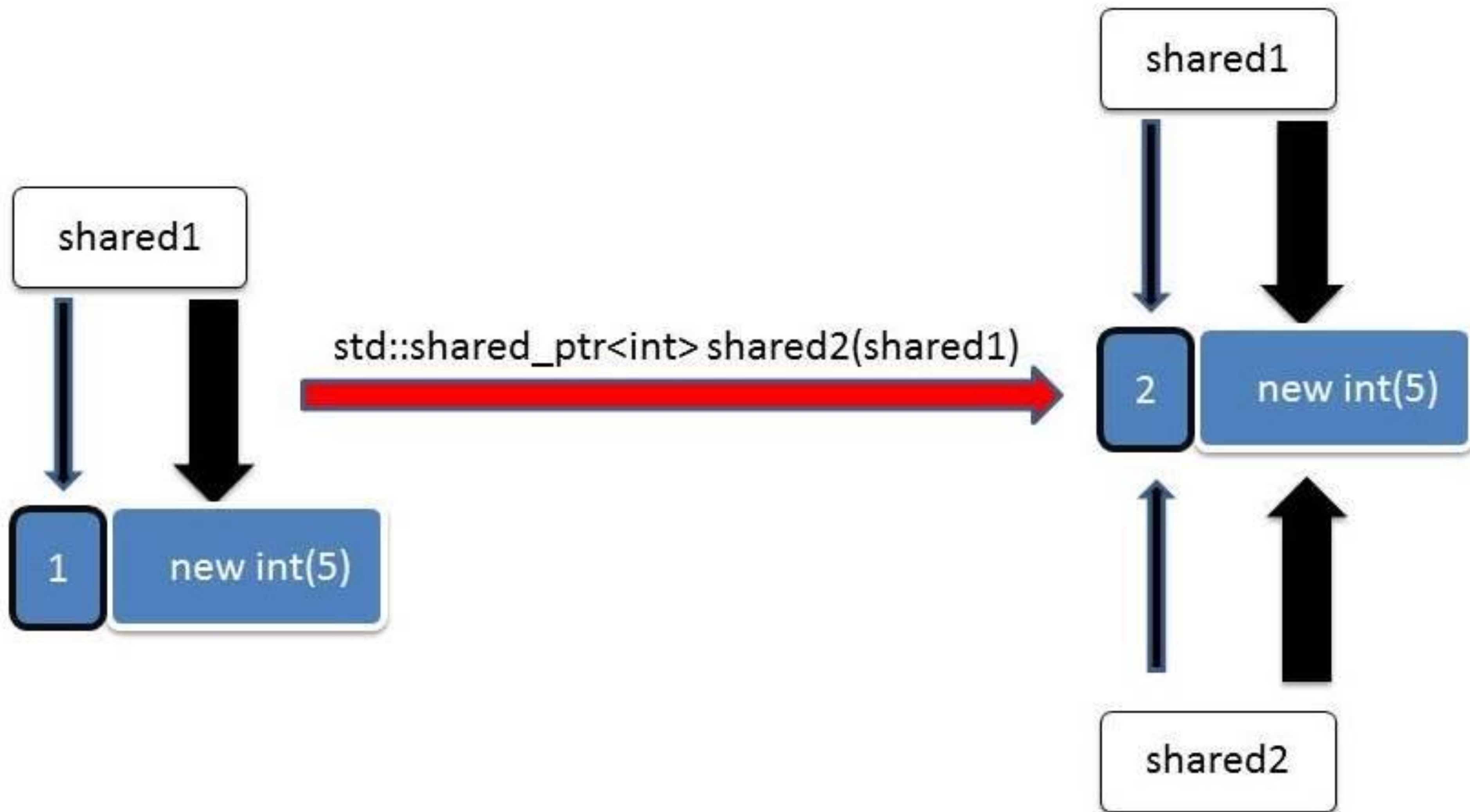
Данный указатель был столь успешным, что его включили в C++ Technical Report 1 и он был доступен в пространстве имен tr1 — std::tr1::shared_ptr<>.

Главное отличие

В отличие от рассмотренных выше указателей, shared_ptr реализует **подсчет ссылок** на ресурс.

Ресурс освободится тогда, когда счетчик ссылок на него будет равен 0.

Как видно, система реализует **одно из основных правил сборщика мусора**.


shared1
1
new int(5)
std::shared_ptr<int> shared2(shared1)
shared1
2
new int(5)
shared2

Пример

```
std::shared_ptr<int> x_ptr(new int(42));
std::shared_ptr<int> y_ptr(new int(13));

// после выполнения данной строчки,

y_ptr = x_ptr;

// ресурс на который указывал ранее y_ptr (int(13)) освободится,
// а на int(42) будут ссылаться оба указателя

std::cout << *x_ptr << "\t" << *y_ptr << std::endl;

// int(42) освободится лишь при уничтожении последнего ссылающегося на него указателя
```

Методы

Также как и unique_ptr, и auto_ptr, данный класс предоставляет методы get() и reset().

```
auto ptr = std::make_shared<Foo>(new Foo);

Foo *foo = ptr.get();
foo->bar();

ptr.reset();
```

Пример

```
#include <memory>
#include <iostream>

int test() {
    std::shared_ptr<MyObject> p1(new MyObject);
    std::shared_ptr<MyObject> p2;
    p2 = p1;
    if (p2)
        std::cout << "Hello, world!\n";
}
```

Теперь и p2 и p1 указывают на один объект, а счетчик ссылок равен 2.
Когда счетчик обнуляется, объект уничтожается.

Передача как параметра

Мы можем передавать этот указатель в функцию:

```
int test(std::shared_ptr<MyObject> p1) {
    // Делаем что-то
}
```

Если передать указатель по ссылке, то счетчик не будет увеличен.

Вы должны быть уверены, что объект MyObject будет жив, пока будет выполняться функция test.

Опасности

При работе с умным указателем, следует опасаться их создания на лету.

Например, следующий код может привести к утечке памяти.

```
someFunction(std::shared_ptr<Foo>(new Foo), getRandomKey());
```

Почему?

В чём же проблема?

Стандарт С++ не определяет порядок вычисления аргументов.

Может случиться так, что
сначала выполнится new Foo, затем getRandomKey() и лишь затем конструктор shared_ptr.

```
someFunction(std::shared_ptr<Foo>(new Foo), getRandomKey());
```

если 3 1 2

Если же функция getRandomKey() бросит исключение, до конструктора shared_ptr дело не дойдет, хотя ресурс (объект Foo) был уже выделен.

Выход есть

Использовать **фабричную** функцию **std::make_shared<>**, которая
создает объект заданного типа и возвращает shared_ptr указывающий на него.

```
someFunction(std::make_shared<Foo>(), getRandomKey());
```

make_shared возвращает shared_ptr, и он является временным объектом

Стандарт C++ четко декларирует, что временные объекты уничтожаются, в случае появления исключения.

new Foo тоже возвращает временный объект
Однако, временным является указатель на выделенный ресурс, и в случае исключения — уничтожится указатель, при этом ресурс останется выделенным.

Когда использовать std::shared_ptr, а когда std::unique_ptr

Если объект нужен только в одном месте, то используйте std::unique_ptr (чтобы защититься от непреднамеренного копирования).

Если объект понадобился в нескольких местах, то — std::shared_ptr.

Более глубокий анализ

std::unique_ptr по своей эффективности очень близок к обычным указателям

Чего нельзя сказать о std::shared_ptr.

Он предоставляет больше возможностей, но за все приходится платить.

Увеличивается и расход памяти, и время доступа.

Однако накладные расходы не столь существенны, поэтому в большинстве приложений разница окажется незаметной.

Итак, smart pointers это хорошо, но есть и минусы

- небольшой оверхед (overhead)

- boilerplate-код

например,

```
shared_ptr<MyNamespace::Object> ptr = shared_ptr<MyNamespace::Object>(new MyNamespace::
                                                        Object(param1, param2, param3))
```

Это частично можно решить при помощи define либо при помощи typedef

- проблема циклических ссылок

Пример

```
struct Widget {
    shared_ptr<Widget> otherWidget;
    …
};

void foo() {
    shared_ptr<Widget> a(new Widget);
    shared_ptr<Widget> b(new Widget);

    a->otherWidget = b;
    // В этой точке у второго объекта счетчик ссылок = 2

    b->otherWidget = a;
    // В этой точке у обоих объектов счетчик ссылок = 2
}
```

Что произойдет при выходе объектов a и b из области определения?

Проблема перекрёстных ссылок

```
struct Widget {
   shared_ptr<Widget> otherWidget;
};

void foo() {
   shared_ptr<Widget> a(new Widget);
   shared_ptr<Widget> b(new Widget);

   a->otherWidget = b;
   // В этой точке у второго объекта счетчик ссылок = 2

   b->otherWidget = a;
   // В этой точке у обоих объектов счетчик ссылок = 2
}
```

В деструкторе уменьшатся ссылки на объекты.

У каждого объекта будет счетчик = 1 (ведь a все еще указывает на b, а b — на a).

Объекты держат друг друга и у приложения нет возможности получить к ним доступ — эти объекты потеряны.

Ещё не всё

Представим себе случай, когда нам нужно передать указатель, не передавая права владения объектом.

Использовать для этой цели shared_ptr нельзя, т.к. его копирование увеличит счётчик ссылок, и в результате происходит разделение прав владения.

Использование обычного указателя также нежелательно, т.к., как уже говорилось в самом начале, невозможно проверить, ссылается указатель на существующий объект или нет.

Решение проблемы

Специально для такого случая предусмотрен ещё один вид умных указателей: weak_ptr.

Этот указатель также как и shared_ptr, начал своё рождение в проекте boost

weak_ptr

weak_ptr — это «слабый», не владеющий экземпляром объекта умный указатель

weak_ptr ссылается на экземпляр, которым владеет shared_ptr, но не разделяет прав владения этим экземпляром

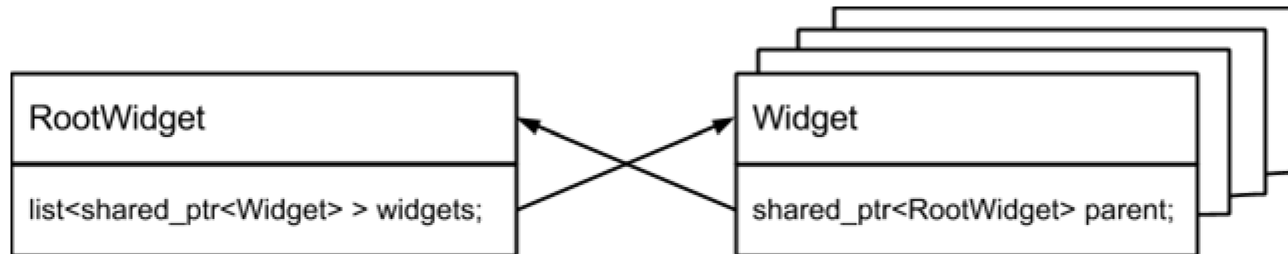
Это гарантирует, что если экземпляр уже уничтожен, мы сможем это надёжно и безопасно проверить

Случай, когда один объект владеет коллекцией других объектов

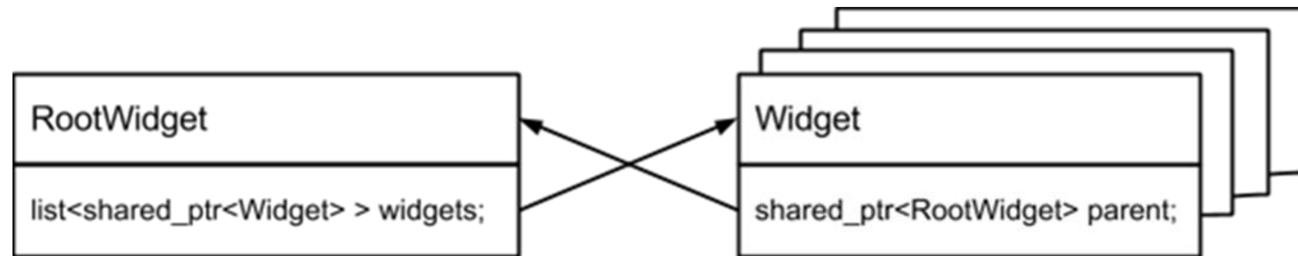
Одним из типичных случаев создания перекрестных ссылок является случай, когда один объект владеет коллекцией других объектов

```
struct RootWidget {
    list<shared_ptr<class Widget> > widgets;
};

struct Widget {
    shared_ptr<class RootWidget> parent;
};
```

При таком устройстве каждый Widget будет препятствовать удалению RootWidget и наоборот.



Решение проблемы перекрестных ссылок



В таком случае нужно ответить на вопрос: «Кто кем владеет?».

Очевидно, что именно RootWidget в данном случае владеет объектами Widget, а не наоборот.

Поэтому модифицировать пример нужно так:

```
struct Widget {
   weak_ptr<class RootWidget> parent;
};
```

Слабые ссылки не препятствуют удалению объекта.

В случае возникновения в коде кольцевых ссылок, используйте weak_ptr для решения проблем.

Преобразование слабых ссылок

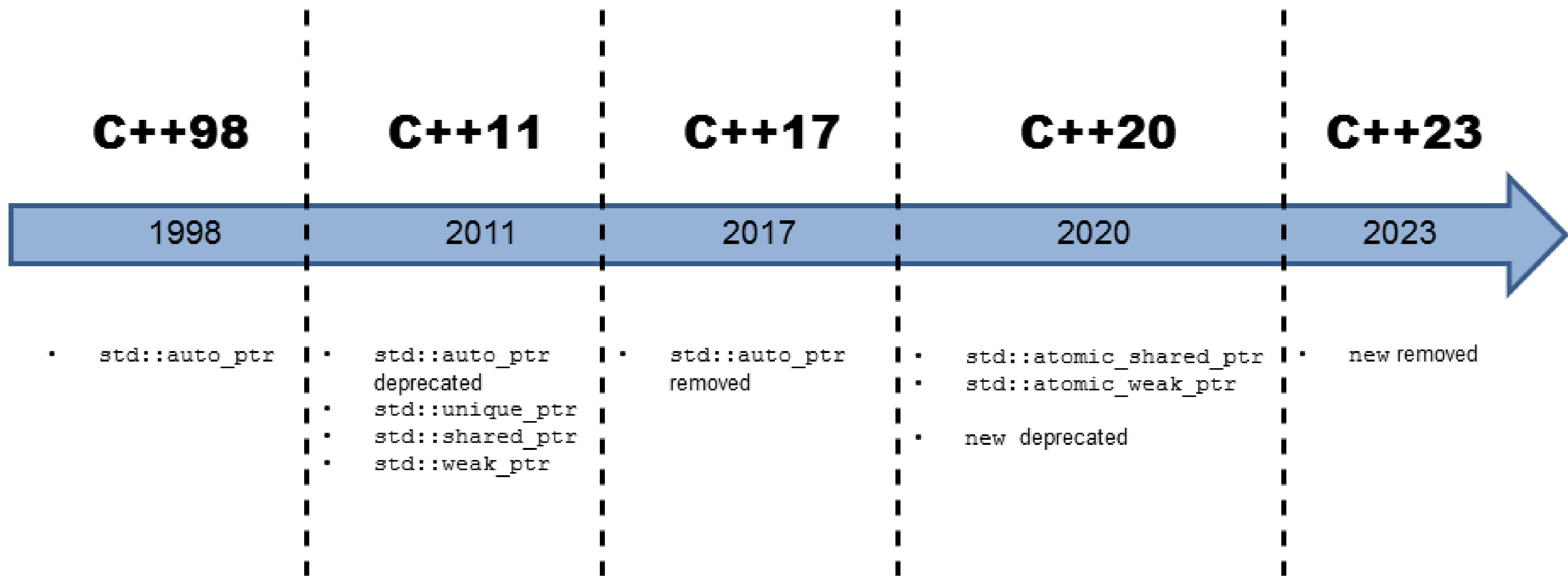
Слабые ссылки могут быть преобразованы в сильные двумя способами

1) **Конструктор shared_ptr**

```
weak_ptr<Widget> w = …;
// В случае, если объект уже удален, в конструкторе shared_ptr будет сгенерировано исключение
shared_ptr<Widget> p( w );
```

2) **Метод lock**

```
weak_ptr<Widget> w = …;
// В случае, если объект уже удален, то p будет пустым указателем
if( shared_ptr<Widget> p = w.lock() ) {
// Объект не был удален – с ним можно работать
}
```


C++98
C++11
C++17
C++20
C++23
1998
2011
2017
2020
2023
std::auto_ptr
std::auto_ptr deprecated
std::unique_ptr
std::shared_ptr
std::weak_ptr
std::auto_ptr removed
std::atomic_shared_ptr
std::atomic_weak_ptr
new deprecated
new removed