NULLPTR

Нулевой указатель можно сформировать:

- 0, который компилятор преобразует (если это возможно) к требуемому типу
- NULL
- **nullptr – C++11**

0

```
void f(int*);
void f(int);
int main()
{
        f(0);  //???
}
```

NULL

```
#include <cstddef> //      #define NULL 0

void f(int n);
void f(int* p);

int main()
{
        f(NULL); //???
}
```

nullptr

(тип std::nullptr_t - <cstddef>)

```
void f(int n);
void f(int* p);

int main()
{
        f(nullptr);  //???
}
```

Правила неявного преобразования nullptr:

- компилятор неявно приводит
  - к типу любого указателя:
    char* pc = nullptr;
    double* pd = <выражение>;
    if(pd == nullptr)…
  - и к типу bool:
    bool b = nullptr; //b==???
- компилятор неявно **НЕ** приводит к любому другому типу:
  - int n = nullptr; //ошибка
  - double d = nullptr; //ошибка

Примеры:

```
void f(int);
int main()
{
        int* pn = NULL; //???
        int n = NULL;  //???

        pn = nullptr;  //???
        n = nullptr;  //???//

        f(nullptr); //???//
        f(NULL); //???
}
```

И даже так:

```
#include <cstddef>
void fn(nullptr_t  np);
void fn(int* p);

int main()
{
        int n = 1;
        fn(&n);
        //fn(0); //неоднозначность
        fn(nullptr);
}
```

ФУНДАМЕНТАЛЬНЫЕ ТИПЫ И СТАНДАРТНЫЕ СУФФИКСЫ И ПРЕФИКСЫ ДЛЯ ЛИТЕРАЛОВ

М. Полубенцева

Проблемы

sizeof(int) ???

sizeof(char) == 1 byte(**== ? bits**)

Стандарт гарантирует только соотношения:

1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)

Добавлены макросы в <cstdint>

До C++11:

CHAR_BIT - количество разрядов в байте

CHAR_MIN – минимальное значение

…

C++11 – добавлены:

SIZE_MAX – максимальное значение переменной типа size_t

WCHAR_MIN – минимальное значение переменной типа wchar_t

…

Проблемы переносимости => псевдонимы C++11
#include <cstdint>

- **int8_t (uint8_t)**
- **int16_t (uint16_t)**
- **int32_t (uint32_t)**
- **int64_t (uint64_t)**

А кроме того:

- **intmax_t (uintmax_t)** – максимально возможное целое
- **intptr_t (uintptr_t)** – целочисленный тип, в который можно принять (reinterpret_cast) значение указателя. Есть/нет-зависит от реализации

Макросы, задающие пределы типов <cstdint >

INTMAX_MIN	Minimum value of intmax_t	$-(2^{63}-1)$, or lower
INTMAX_MAX	Maximum value of intmax_t	$2^{63}-1$, or higher
UINTMAX_MAX	Maximum value of uintmax_t	$2^{64}-1$, or higher
…	…	…

Примеры:

- size_t n1 = sizeof(**int**); //???
- size_t n2 = sizeof(**int32_t**); //???

C++11 - «Длинные» целые типы и литералы long long

- целое, размером не менее 64 битов
- суффикс для литералов – LL

```
long long l = 123456789123LL;
```

C++14 – двоичные литералы

- Префикс 0B или 0b

0b10111000

C++14 - для увеличения читабельности литералов – **разделители разрядов**

```
long long  n = 8'921'111'22'33;

int m = 0b0100'1100'0110;
```

C++11 – возможность проверки <type_traits>

- is_void - проверяет, является ли тип void
- is_integral
- is_floating_point
- is_array
- is_enum
- …

Пример

```
template<typename T> void fTypeTraits(const T& t){
if (std::is_integral<T>::value) {…}
}

int main()
{
        fTypeTraits(55);
        fTypeTraits(5.5);
}
```

По аналогии:

- is_fundamental
- is_arithmetic
- is_scalar

Свойства типов
#include <type_traits>

- is_const
- is_trivial - проверка: класс/структура с тривиальными данными + тривиальное копирование (автоматически сгенерированные операции)
- is_empty – проверка: в классе/структуре (без статических данных) нет данных

Пример is_trivial

```
struct A {     int m; }; //тривиальный
struct A1 {     int m=0; }; //нетривиальный

struct B {     B() {} };

template<typename T> void fIsTrivial(const T& t){
        if(std::is_trivial<T>::value) {…}
}
int main(){
        fIsTrivial(A());
        fIsTrivial(B());
}
```

C++11 **Отношения типов**
#include <type_traits>

- is_same () проверяет два типа одинаковы
- is_base_of checks if a type is derived from the other type
- is_convertible проверяет, является ли тип может быть преобразован в другой тип

Пример:

```
bool b1 =
        std::is_convertible<double, int>::value;

bool b2 =
        std::is_convertible< int, double>::value;
```

Пример

```
class A {};
 class B: public A {};
 class C {};

   bool b2a = std::is_convertible<B*, A*>::value;
   bool a2b = std::is_convertible<A*, B*>::value;
   bool b2c = std::is_convertible<B*, C*>::value;
```

не поддерживается VS 13, VS 15 - ОК

СИМВОЛЫ. ПОДДЕРЖКА КОДИРОВОК UNICODE

C++11 поддерживает:

- + к const char и const wchar_t литералам
- UTF-8
- UTF-16
- UTF-32

=> добавлены типы:

- char16_t – для хранения UTF-16 символов
- char32_t – для хранения UTF-32 символов

Префиксы:

- const char[]
  u8"UTF-8 string"
- const char16_t[]
  u"UTF-16 string"
- const char32_t[]
  U"UTF-32 string"

Примеры

```
const char16_t * pstr = u"UTF-16 string";

std::basic_string<char16_t> ustr =
                              u"UTF-16 string";
```

ПОЛЬЗОВАТЕЛЬСКИЕ ЛИТЕРАЛЫ

Пользовательские литералы

Позволяют **вычислять** значения (как базового, так и любого пользовательского типа) с помощью:

- **литералов** (!категория ограничена – целые, плавающие, символы и строки),
- задаваемых пользователем **суффиксов**
- и literal operators - **operator " "**

Замечание: в большинстве случаев вычисление возможно на **этапе компиляции**!!!

Хотелось бы:

```
{
//программист задает интервал времени в часах, а программа оперирует значениями в секундах
        int interval_sec = 24_hours;

// или хочется вывести на консоль 100 км
        std::cout<<100_km;
 }
```

Замечание:

Для пользовательских литералов можно задавать **только** суффиксы!

(префиксы, например **hours_24**, задать невозможно)

Специфика формирования суффиксов

- идентификатор, сформированный согласно правилам C++
- рекомендуется (необязательно) начинать с подчерка (так как без подчерка уже используются + зарезервированы на светлое будущее для использования стандартной библиотекой)

Синтаксис (1):

[constexpr] тип_возвращаемого_значения
operator"" ***суффикс***
(<тип_параметра> ***parameter***)
{
 …
 return ***выражение***;
};
, где тип параметра может быть только одним из:

unsigned long long
long double
char
const char*

Синтаксис (2):

[constexpr]

тип_возвращаемого_значения

operator"" ***суффикс***

(const char* , size_t) ;

//const wchar_t*, const char16_t*, const char32_t*

Что делает компилятор, встречая пользовательский литерал:

- если суффикс не совпадает ни с одним из «стандартных» (f, LL…)
- ищет функцию с именем **operator"" суффикс**
- если нашел подходящую форму перегрузки, «отрезает суффикс»
- и вызывает оператор с оставшимся значением

Примеры задания пользовательских литералов

```
unsigned long long operator""_hours(
                                    unsigned long long n)
{
        return n*60*60;
}
int main()
{
        unsigned long long interval_sec = 24_hours;
                                                //в секундах
}
```

Примеры задания пользовательских литералов

???

```
int main()
{
   std::cout << 100_km; //нужно вывести строчку 100 км
   std::cout << 200_km; //нужно вывести строчку 200 км
}
```

Специфика задания пользовательских литералов

- нельзя переопределить поведение встроенных литералов
- нельзя расширить синтаксис литералов (тип параметра ограничен)
- литеральный оператор может принимать значение в кавычках (в качестве строки) или без них.
- для использования литерала без кавычек достаточно определить оператор, принимающий единственный аргумент типа **const char***:

Пример специфического использования пользовательских литералов

```
void operator"" _printDecimal (unsigned long long n)
{
        std::cout << n;
}
int main()
{
        0x1122eeff_printDecimal;
                                //вывод: 287502079
}
```

Специфика задания пользовательских литералов

Если для данного суффикса определен только один оператор вида:
тип **operator"" суффикс(const char* str)**

явно задавать строку при вызове не требуется!

```
void operator"" _printStr(const char* str)
{
        std::cout << str;
}
int main()
{
        0x1122eeff_printStr; //вывод: 0x1122eeff
}
```

Пример:

<тип> operator "" _x(**unsigned long long**);

<тип> operator "" _y(**const char***);

```
1234_x;    // вызов operator "" _x(1234);
1234_y;    // вызов operator "" _y("1234");
```

Специфика перегрузки пользовательских литералов

```
void operator""_print(const char* str, size_t n)
{std::cout << n <<" symbols in "<< str << std::endl; }

void operator""_print(unsigned long long n)
{ std::cout << n << std::endl;  }

int main()
{
        "0x1122eeff"_print; //(const char* str, size_t n) n=10 (без учета '\0')
        0x1122eeff_print; //(unsigned long long n)
}
```

Для второй формы сигнатура выбирается в зависимости от типа строки:

"123"_s; // operator "" _s(const char* str, size_t size);

u8"123"_s; // operator "" _s(const char* str, size_t size);

L"123"_s; // operator "" _s (const wchar_t* str, size_t size);

u"123"_s; // operator "" _s(const char16_t* str, size_t size);

U"123"_s; // operator "" _s(const char32_t* str, size_t size);

C++14 добавляет следующие стандартные суффиксы (и соответствующие operator"")

- «s» для создания различных std::basic_string типов.
- «h», «min», «s», «ms», «us» и «ns» для создания соответствующих временных интервалов std::chrono::duration.
- «if», «i», «il» для комплексных чисел

Замечание: все стандартные литеральные операторы заключены в соответствующие namespace

Пример использования стандартных литеральных операторов – C++14

```
using namespace std::string_literals;
string str = "hello world"s;

using namespace std::literals::chrono_literals;
auto interval = 60s; //в секундах
```

Замечание: два литерала «s» не конфликтуют, поскольку строковой литерал работает только над строками, а секундный над числами

C++14 Variadic template пользовательский литерал!!!

template <**char...**> <тип>

operator"" _<суффикс>();

Перегрузка пользовательских литералов

Порядок выбора компилятором:

- operator "" _x (unsigned long long) или operator "" _x (long double)`
- operator "" _x (const char* raw)
- operator "" _x <'c1', 'c2', ... 'cn'>

Generalized constant expressions

СПЕЦИФИКАТОР **CONSTEXPR**

Спецификатор **constexpr** можно использовать для задания:

- константных «переменных»
- функций
- и даже объектов (при этом ограничения на конструктор)

Важно: значения должны быть известны на этапе компиляции!!!

constexpr-функция

```
constexpr int sum (int a, int b) { return a + b; }
int main()
{
        constexpr int n = sum (5, 12); // значение будет
                                       посчитано на этапе компиляции
        int x=1, y=2;
        int m = sum(x,y); //полноценный вызов функции
}
```

Пользовательские литералы и constexpr

```
constexpr unsigned long long operator""_hours(
        unsigned long long n)
{       return n * 60 * 60;          }

constexpr std::string operator""_km
                        (unsigned long long n) //ошибка!!!
{
        std::string s = std::to_string(n);
        return s + " km";
}
```

Специфика constexpr – функции

- C++11 - должна состоять из единственной инструкции
  return <выражение>;
  C++14 ограничение до некоторой степени снято!
- constexpr – функции по умолчанию являются встраиваемыми (inline) =>
- определение – где?

Пример использования

client.cpp

```
#include “server.h”

int main()
{
        const int a=3, b=5;
        int ar1[sum(a, b)];//???

        int n=4, m=5;
        int ar2[sum(n, m)];//???
        …
}
```

server.h

```
constexpr int sum(int x, int y)
 { return x + y;  }
```

Ограничения constexpr-функции

- тело функции не должно содержать действий, которые нельзя выполнить на этапе компиляции
- функция должна обязательно возвращать значение
- тело constexpr-функции должно быть известно компилятору на момент «вызова»

Специфика:

- если значения параметров возможно вычислить на этапе компиляции, то возвращаемое значение также должно вычисляться на этапе компиляции
- если значение хотя бы одного параметра будет неизвестно на этапе компиляции, то будет сгенерирован полноценный вызов (ошибки не будет)

Пример constexpr-функции

```
//сравнение символов
constexpr int my_char_cmp( char c1, char c2 ) {
        return (c1 == c2) ? 0 : ((c1 < c2) ?  -1 : 1);
}

constexpr int s = my_char_cmp('W','A');
```

Пример C++11 - сравнение строк

// Только посредством рекурсии! => много накладных расходов!

???

Дополнительные возможности constexpr-функций в C++14

- определения локальных переменных
  - должны быть проинициализированы
  - не могут быть static
  - не могут быть thread_local
- в теле функции допускается модификация локальных переменных
- можно использовать инструкции:
  if, switch, for, while, do-while (кроме goto)

C++14 Переписываем сравнение строк без рекурсии (VS15 пока не работает!!!)

```
constexpr int my_strcmp_14
                                (const char* s1, const char* s2)
{
        int i = 0;
        for (;  s1[i] == s2[i] && s1[i]; ++i){}
        return s1[i] -s2[i];
}
```

constexpr-переменная

- похожа на «простую константу»
- инициализирующее значение должно быть известно на этапе компиляции

Отличия constexpr и const

```
int x = 2;
constexpr int i = x; //ошибка
const int j = x; //??
```

constexpr-переменная

```
constexpr int sum(int a, int b)  { return a + b;  }

int main()
{
        int k = sum(4, 5);
        int ar1[k];//???

        constexpr int n = sum(4, 5);
        int ar2[n];//???

        //эквивалентно
        const int m = sum(4, 5);
        int ar3[m];//???
}
```

Отличия constexpr и const

constexpr применяется к declaration в целом и не является частью типа:

constexpr char *p1 = "ABC"; //char* const

const char*p2 = "ABC"; //const char*

constexpr – класс

```
template <typename T>
class ConstValue
{
        const T m_t;
public:
constexpr ConstValue(const T& t) : m_t(t) { } // Все методы должны быть constexpr – C++11
constexpr T get() const { return m_t; }
};

ConstValue<int> c(5);
int x[ConstValue<int>(3).get()];
```

Ограничения на конструктор в constexpr - классе

- Все нестатические члены класса и члены базовых классов должны быть проинициализированы
  - в конструкторе, используя списки инициализации
  - или инициализацией членов класса при объявлении
  - инициализирующие выражения должны содержать только литералы или constexpr-переменные и constexpr-функции

constexpr и пользовательские литералы

```
constexpr int operator""_modul(char x){
return (x < 0) ? -x : x;
}

constexpr char c1 = 'f'_modul;
constexpr char c2 = 'ф'_modul;
```

RAW STRING LITERALS

Префикс «R»

предписывает компилятору не обрабатывать комбинации

\<символ>

, заключенные в ( ) как escape-последовательность

Пример

```
std::string s1("C:\\myDir\\myFile.txt");
std::cout << s1 << std::endl; // C:\myDir\myFile.txt

std::string s2 = R"(C:\myDir\myFile.txt)";
std::cout << s2 << std::endl; // C:\myDir\myFile.txt

//но!
std::string s3(R"(C:\\myDir\\myFile.txt)");
std::cout << s3 << std::endl; // C:\\myDir\\myFile.txt
```

Специфика:

В некоторых случаях использование символа ‘)’ может привести к ошибке:

char ar[] = R"(Example R"(Raw string)" string)";

=> было введено понятие delimeter

Delimeter

- -это строка длиной до 16 символов, включая пустую строку
- delimeter не может содержать пробелы, управляющие символы, '(', ')', или символ '\‘
- использование в "сырых" строковых литералах:
  R"delimiter**(**любая_строка**)**delimiter**"**

Использование delimeter

```
char ar[] =
        R"aaa(Example R"(Raw string)" string)aaa";

// Example R"(Raw string)" string
```

Примеры использования сырых строковых литералов

```
char ar1[] = "abc(\')abc";
size_t n1= sizeof(ar1);          //???

char ar2[] = R"abc(\')abc";
size_t n2 = sizeof(ar2);         //???

const char* pc2 = R"*(\')*";
const wchar_t* pw = LR"(\')";
```

Полезные добавления в класс string

ФУНКЦИИ ДЛЯ РАБОТЫ СО СТРОКАМИ (STRING)

М. Полубенцева

std::basic_string - C++11 – добавлены:

- конструкторы

basic_string(basic_string&& other)

basic_string(std::initializer_list<CharT> init,
const Allocator& alloc = Allocator());

- operator=

basic_string& operator=(basic_string&& str);

basic_string& operator=(std::initializer_list
<CharT> ilist);

До стандарта C++11

- atoi
- atol
- atoll

C++11: преобразование string в целое. Функции:

int std::**stoi**(const std::string& str,
size_t *pos = 0, int base = 10); //*pos = индекс первого непреобразованного символа

long std::**stol**(const std::string& str,
size_t *pos = 0, int base = 10);

long long std::**stoll**(const std::string& str,
size_t *pos = 0, int base = 10);

C++11: преобразование string в значение. Генерируемые исключения:

- **std::invalid_argument**, если преобразование не может быть выполнено
- **std::out_of_range**, если преобразованное значение будет выходить за границы диапазона значений типа результата

Пример:

```
std::string str = "f1"; // или "0xf1";
int myint = stoi(str,0,16);

try{
        std::string str = "qwerty";
        myint = stoi(str);
}
catch (std::invalid_argument& i)
{
std::cout << i.what(); //'invalid stoi argument'
}
```

C++11: преобразование string в плавающее

- stof
- stod
- stold

C++11 преобразование в string

(идентично std::sprintf с соответствующим спецификатором %)

- std::string **to_string**(int);
- std::string **to_string**(long);
- std::string **to_string**(long long);
- std::string **to_string**(unsigned int);
- std::string **to_string**(unsigned long);
- std::string **to_string**(unsigned long long);
- std::string **to_string**(float);
- std::string **to_string**(double);
- std::string **to_string**(long double);

C++11 преобразование в wstring (идентично std::swprintf с соответствующим спецификатором %)

- to_wstring аналогично to_string

Smart Pointers

Интеллектуальные указатели

Эпиграф:

Указатель – это бумеранг…

Проблемы.
«Голые» указатели

- при неаккуратном использовании могут порождать **утечки памяти**
- **разыменование нулевого указателя** == ошибка времени выполнения
- **попытка удалить уже удаленный объект** (вызов delete для недействительного указателя) == в лучшем случае ошибка времени выполнения (в худшем в Release версии – трудно выявляемые логические ошибки)

А также:

- **Время жизни** указателя и указываемого объекта никак не связаны => проблема «недействительных» указателей :
  - и тот, и другой могут быть локальными, динамическими, статическими
  - проблемы - даже, если оба локальные, но имеют разное время жизни

А также:

- Указатели, ссылающиеся на один и тот же объект, никак не связаны между собой. Это тоже создаёт проблему «недействительных» указателей – указателей, ссылающихся на освобождённые или перемещённые объекты.
- Нет никакой возможности во время выполнения проверить, указывает ли указатель на корректные данные, либо «в никуда».
- Указатель на единичный объект и указатель на массив объектов никак не отличаются друг от друга.

Иллюстрация проблем:

```
{
        MyObject o;
        MyObject* pO = new MyObject;
        //используем o и pO
        //delete забыли
}//???
```

Иллюстрация проблем:

```
{
        MyObject* p = nullptr;
        {
                MyObject localObject;
               p = & localObject;
        }
        p //???
}
```

Иллюстрация проблем:

```
// А так?
try{
        MyObject* p = new MyObject;
        throw “error”;
        delete p;
}catch(…)
{…}
```

Защита от утечек ресурсов при генерации исключения:

```
MyObject* p = nullptr;
try {
        p = new SomeClass;
        throw “error”;
        delete p;
}
catch (...)
{ delete p; };
```

Код разрастается…

```
void f()
{
        MyObject* p = new MyObject;
        if(усл.1) {delete p; return;}
        …
        if(усл.2) {delete p; return;}
        …
        delete p;
}
```

Управление ресурсами посредством локальных объектов (идиома **RAII** - ***R**esource **A**cquisition **I**s **I**nitialialization*)

Цель: при создании объекта захват ресурса, при уничтожении – освобождение =>

- В конструкторе – получение доступа к ресурсу (выделение памяти, открытие файла, установление сетевого соединения…)
- В деструкторе – освобождение ресурса (освобождение памяти, закрытие файла, завершение соединения…)

Важно!

Создание оберток для указателей (и соответствующие приемы)

- не являются панацеей (полностью не освобождают программиста от «**надо думать**»)
- но при корректном использовании облегчают ему (программисту) жизнь

Определение smart pointer

- smart pointer (умный указатель) — это класс, имитирующий интерфейс обычного указателя и добавляющий некую новую функциональность (например, проверку границ при доступе или удаление неиспользуемого «никем другим» объекта,…)

Эмуляция smart pointer (MyUniquePtr)

```
template <typename T> class smart_pointer {
        T *m_obj;
public:
        smart_pointer(T *obj) : m_obj(obj) { }
        ~smart_pointer() { delete m_obj; }
        //???
//Для удобства:
T* operator->() { return m_obj; } //так как это обертка для указателя
T& operator* () { return *m_obj; }
};
```

Целевой класс

```
class A{
        int m_a;
public:
        void something(){}
        A(int a) :m_a(a){}
friend std::ostream& operator<<
  (std::ostream& os, const A& a){ os << a.m_a; return os; }
};
```

Использование smart_pointer :

```
{
        smart_pointer<A> pA(new A(5));
        pA->something(); //???
        std::cout << *pA << std::endl; //???
} //???
```

Использование. Важно!

- использование smart pointer–ов не предполагает динамического создания самих оберток. Почему ?
- обычно обертка хранит результат непосредственного вызова оператора new. Почему?
- **конкретно в нашей реализации** нельзя «заворачивать» в smart_pointer указатель на локальный или статический объект. Почему?

???

```
{
   smart_pointer<A>* pPointer = new
         smart_pointer<A>(new A(5));

   A* p = new A(33);
   smart_pointer<A> pA1(p);
   smart_pointer<A> pA2(p);

   A a(55);
   smart_pointer<A> pA3(&a);

}
```

Проблема:

```
{
        smart_pointer<A> pA1(new A(5));
        smart_pointer<A> pA2 = pA1;
 //???
} //???
```

Решение?

Решение

```
template <typename T> class smart_pointer {
        T *m_obj;
public:
  smart_pointer(T *obj) : m_obj(obj) { }
  ~smart_pointer() { delete m_obj; }
  smart_pointer(const smart_pointer&) = delete;
  smart_pointer(smart_pointer&& other) {???}
  …
};
```

Демонстрация решения:

```
{
   smart_pointer<A> pA1(new A(5));
   smart_pointer<A> pA2 = pA1; //???
   smart_pointer<A> pA3 = //???

   pA1 = pA3; //???
   pA1 = //???
} //???
```

Демонстрация решения:

```
void  f1(smart_pointer<A> p){}
smart_pointer<A> f2()
{smart_pointer<A> p(new A(33)); return p; /*???*/}

int main(){
   smart_pointer<A> pA1(new A(5));
   f1(pA1); //???
   smart_pointer<A> pA2 = f2(); //???
   pA1 = f2(); //???
} //???
```

Отличия? Что лучше? Проблемы?

```
void  f1(smart_pointer<A> );
void  f2(smart_pointer<A>& );

int main(){
   smart_pointer<A> pA1(new A(1));
   f1(pA1); //???
   //pA1 ???
   smart_pointer<A> pA2 (new A(2)); //???
  f2(pA2); //???
  //pA2 ???
}
```

C++11
#include <memory>

Стандартная библиотека предоставляет **smart pointers**, которые способствуют созданию программ:

- без утечек ресурсов (памяти,… и не только!)
- безопасных (с точки зрения исключений)

C++11 – типы smart pointers

- **unique_ptr** (вместо auto_ptr - deprecated) – гарантирует только ОДНОГО владельца для обернутого указателя => указатель может только сменить владельца (копировать запрещено, можно передавать другому)
- **shared_ptr** - класс с подсчетом ссылок => реализует концепцию совместного владения (первый создает, последний удаляет)
- **weak_ptr** - предоставляет (для временного пользования) доступ к объекту, завернутому в shared_ptr (не участвует в подсчете ссылок, не имеет права создавать и удалять)

STD::UNIQUE_PTR

std::unique_ptr

```
template<  //обертка для одиночного объекта
    class T, //целевой тип
    class Deleter = std::default_delete<T>
> class unique_ptr;
template < //обертка для массива объектов
    class T,
    class Deleter
> class unique_ptr<T[],Deleter>; //operator[]
```

Типичные случаи применения std::unique_ptr :

- обеспечение безопасности исключений для классов и функций, которые управляют объектами с динамическим временем жизни, гарантируя удаление в случае:
  нормального завершения
  и завершения по исключению
- передача владения динамически созданным объектом в функцию
- получение владения динамически созданным объектом из функций
- в качестве типа элемента в контейнерах, поддерживающих семантику перемещения, таких как std::vector, которые хранят указатели на динамически выделенные объекты (например, если желательно полиморфное поведение)

Демонстрация использования std::**unique_ptr**

```
{//одиночный объект
        std::unique_ptr<A> ptrA1(new A(55));
        ptrA1->something();
        A a = *ptrA1;

        std::unique_ptr<A> ptrA2 = ptrA1; //???
        std::unique_ptr<A> ptrA3 = std::move(ptrA1); //???
        ptrA1 = ptrA3 //???
        ptrA1 = std::move(ptrA3);  //???
}  //???
```

Альтернативный способ инициализации – шаблон **make_unique**() C++14

```
template< class T, class... Args > unique_ptr<T>
        make_unique(Args&&... args)
{
    return unique_ptr(new
                      T(std::forward(args)...));
}
```

Замечание: + перегруженные варианты для массива

std::make_unique() C++14

```
std::unique_ptr<A> ptrA =
                    std::make_unique<A>(55);
```

- явно принимает параметры, предназначенные конструктору целевого объекта
- неявно принимает адрес «под возвращаемое значение»
- динамически создает целевой объект
- по неявно принятому адресу конструирует объект unique_ptr

Рекомендация

```
//вместо
std::unique_ptr<A> ptrA1(new A(55));

//а особенно вместо
A* p = new A(55);
std::unique_ptr<A> ptrA2(p);

//рекомендуется
std::unique_ptr<A> ptrA3 =
                    std::make_unique<A>(55);
```

Получение указателя из обертки

- pointer **get()** const;
- pointer **release();** //возвращает указатель, а переменную объекта обнуляет
  => отключает обертку от указателя
  => с возвращенным указателем программист должен разбираться самостоятельно

Уничтожение целевого объекта происходит :

- автоматически при завершении времени жизни (самый желательный способ)
- автоматически в случае генерации исключения при «раскрутке стека»
- при вызове метода unique_ptr::**reset**()
- при выполнении move – присваивания

Замена указателя в обертке: std::unique_ptr::**reset()**

void **reset**(pointer ptr = pointer());

- для текущего указателя вызывает функцию-deleter
- в переменной класса запоминает указатель, полученный в качестве параметра

Деструктор unique_ptr:

- если хранящийся указатель==nullptr, деструктор ничего не делает
- в противном случае вызывает delete – функцию (по умолчанию или пользовательскую)
  Замечание: C++14 – по умолчанию использует в качестве функционального объекта шаблон структуры:
  - std::default_delete – для одиночного объекта (вызывается operator **delete**)
  - std::default_delete <T[]> - для массива (вызывается operator **delete[]**)

delete – функция

- заданная по умолчанию – вызывает:
  - оператор delete для одиночного объекта
  - оператор delete[] для массива
- пользовательская – предусмотренные программистом специфические действия

Пользовательская delete-функция. Пример использования unique_ptr **без динамического выделения памяти**

```
{
  unique_ptr<FILE, int(*)(FILE*)>
        my_file(fopen("test.txt", "w"), &fclose);
//или:
  unique_ptr<FILE, decltype(&fclose)>
        my_file(fopen("test.txt", "w"), &fclose);
  char ar[] = "test" ;
  if(my_file) fwrite(ar, sizeof(ar)/sizeof(char),
                sizeof(char), my_file.get());
}
//Когда будет закрыт файл?
// «Кто» вызовет fclose?
```

Пример использования unique_ptr в качестве обертки для массива

```
{
  int n = 5;
  std::unique_ptr<A[] > p(new A[n]);
  auto p1 = std::make_unique<A[]>(n); //unique_ptr<A[], std::default_delete<A[]>>

  for (int i = 0; i < n; ++i)
  {
        p[i] = A(i);
        std::cout << p[i] << std::endl;
  }
} //???
```

Проблема: о количестве элементов в массиве знает только программист! =>

Использование лямбда в качестве delete-функции

```
{
  std::unique_ptr<FILE, void(*)(FILE*)>
                        my_file(fopen("test.txt", "w"),
                  [](FILE* pf) {fclose(pf); });

  char ar[] = "testLambda";
  if (my_file)
        fwrite(ar, sizeof(ar) / sizeof(char), sizeof(char),
                                                  my_file.get());
}
```

STD::SHARED_PTR

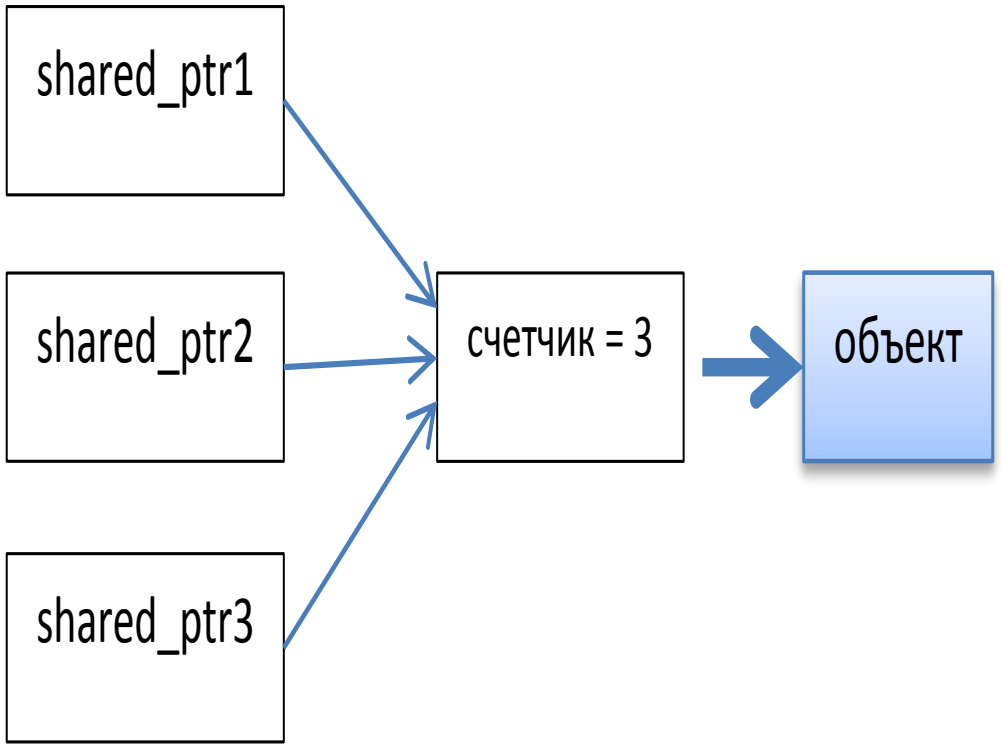
std::shared_ptr

Предоставляет возможность одновременно обращаться к объекту нескольким «пользователям».

Гарантирует: пока существует хотя бы один пользователь ресурса/объекта, объект уничтожен НЕ будет!

- класс-обертка для указателя
- класс с подсчетом ссылок

Класс с подсчетом ссылок



Для корректного использования shared_ptr нужно учитывать:

- перекрестные ссылки;
- чем опасны безымянные shared_ptr;
- какие опасности подстерегают при использовании shared_ptr в многопоточной среде;
- о чем важно помнить, создавая свою собственную delete - функцию для shared_ptr;
- какие существуют особенности использования шаблона enable_shared_from_this.

Основная функциональность std::shared_ptr

- конструктор копирования
- operator= (несколько перегруженных вариантов)

- use_count – число владельцев
- unique – проверка: текущий владелец – единственный?

- operator==
- operator->
- operator*
- operator bool – если указатель == nullptr, то false

Операторы явного приведения типа

- std::static_pointer_cast
- std::dynamic_pointer_cast
- std::const_pointer_cast

Применяют соответственно static_cast, dynamic_cast, const_cast к хранящемуся указателю

Подсчет ссылок происходит только при копировании и присваивании!

```
shared_ptr<string> sh1(new string("A")); //count=1
shared_ptr<string> sh2(new string("B")); //count=1
{
shared_ptr<string> sh3 = sh2; //sh3 : count=2 ,
                              sh2 : count=2 , sh1 : count=1
sh3 = sh1; // sh2 : count=1 , sh3 : count=2 , sh1 : count=2
}//???
// sh2 : count=??? ,  sh1 : count=???
```

Инициализация

```
{
std::shared_ptr<std::string> sh1(new std::string("AAA")); //ОК
//std::shared_ptr<std::string> sh2 = new std::string("AAA"); //explicit!
std::shared_ptr<std::string> sh3 { new std::string("AAA") };
                                                    //OK
//std::shared_ptr<std::string> sh4 = { new std::string("AAA") };//explicit!

//Предпочтительнее
std::shared_ptr<std::string> sh5 =
                    std::make_shared<std::string>("qqq");
} //???
```

operator=

```
shared_ptr<int> sh1(new int(1));
{
   shared_ptr<double> sh4(new double(1.1));
   //sh4 = sh1; //ошибка - типы разные
   int n = 2;

  shared_ptr<int> sh2(&n, [](int* p) {/*"Do nothing"*/});
  shared_ptr<int> sh3(&n, [](int* p) {/*"Do nothing"*/});
  sh1 = sh2; //delete- функции разные! - ???
} //sh2 ???
```

Смена владельца:

void **reset**(); //???

template< class Y > void reset(Y* ptr);

//deleter остается прежним

Смена владельца:

```
std::shared_ptr<std::string> sh ;
//или
std::shared_ptr<std::string> sh1(new std::string("AAA"));

sh.reset(new std::string("BBB")); //???
sh1.reset(new std::string(“CCC")); //???
sh1.reset(); //???
```

Смена владельца с заменой delete-функции:

```
template< class Y, class Deleter >
        void reset( Y* ptr, Deleter d );
```

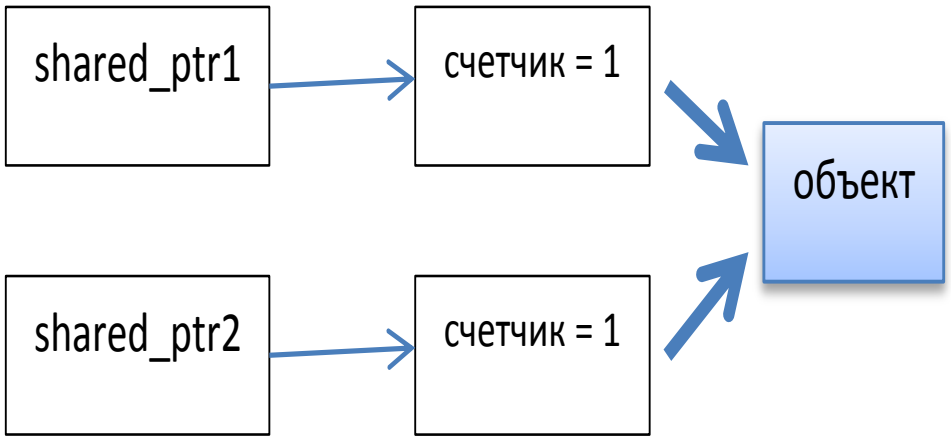
Совместное использование

Важно!

для совместного использования следует оперировать ТОЛЬКО объектами shared_ptr, а не указателями на целевой объект.

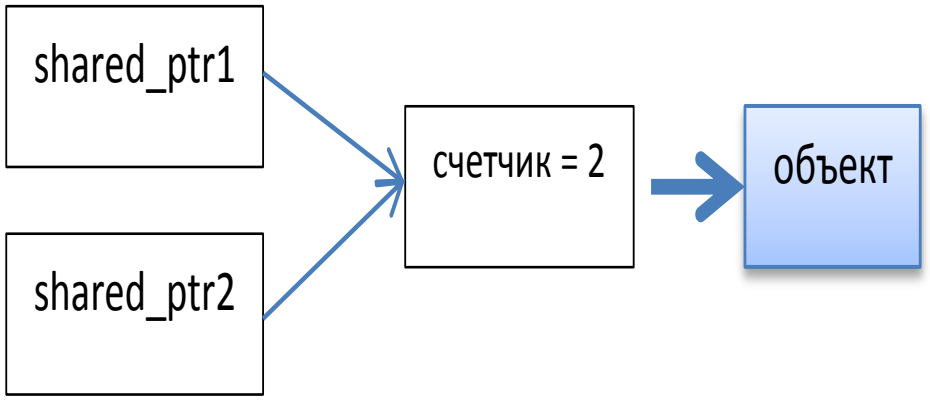
Совместное использование. Пример ошибочного использования

```
A* p = new A;
shared_ptr<A> sh1(p); //count==?
shared_ptr<A> sh2(p); //count==?
```



Совместное использование. Пример корректного использования

```
shared_ptr<A> sh1(new A); //count==?
shared_ptr<A> sh2(sh1); //count==?
```



Проблема анонимных объектов shared_ptr

```
void f(shared_ptr<A> , int);
int excFun(){ throw “error”;…}

int main()
{
        f(shared_ptr<A>(new A(5)), excFun());
}
```

Гарантированно: вызов функции будет последним действием, а объект shared_ptr будет создан после вызова new

Нет гарантии, что конструирование shared_ptr будет завершено раньше, чем вызов excFun()

Рекомендация по решению проблемы анонимных объектов shared_ptr:

```
void f(shared_ptr<A> , int);
int excFun(){ throw “error”;…}

int main()
{
        shared_ptr<A> ptr(new A(5));
        f( ptr, excFun() );
//или
        f( make_shared<A>(5), excFun() );
}
```

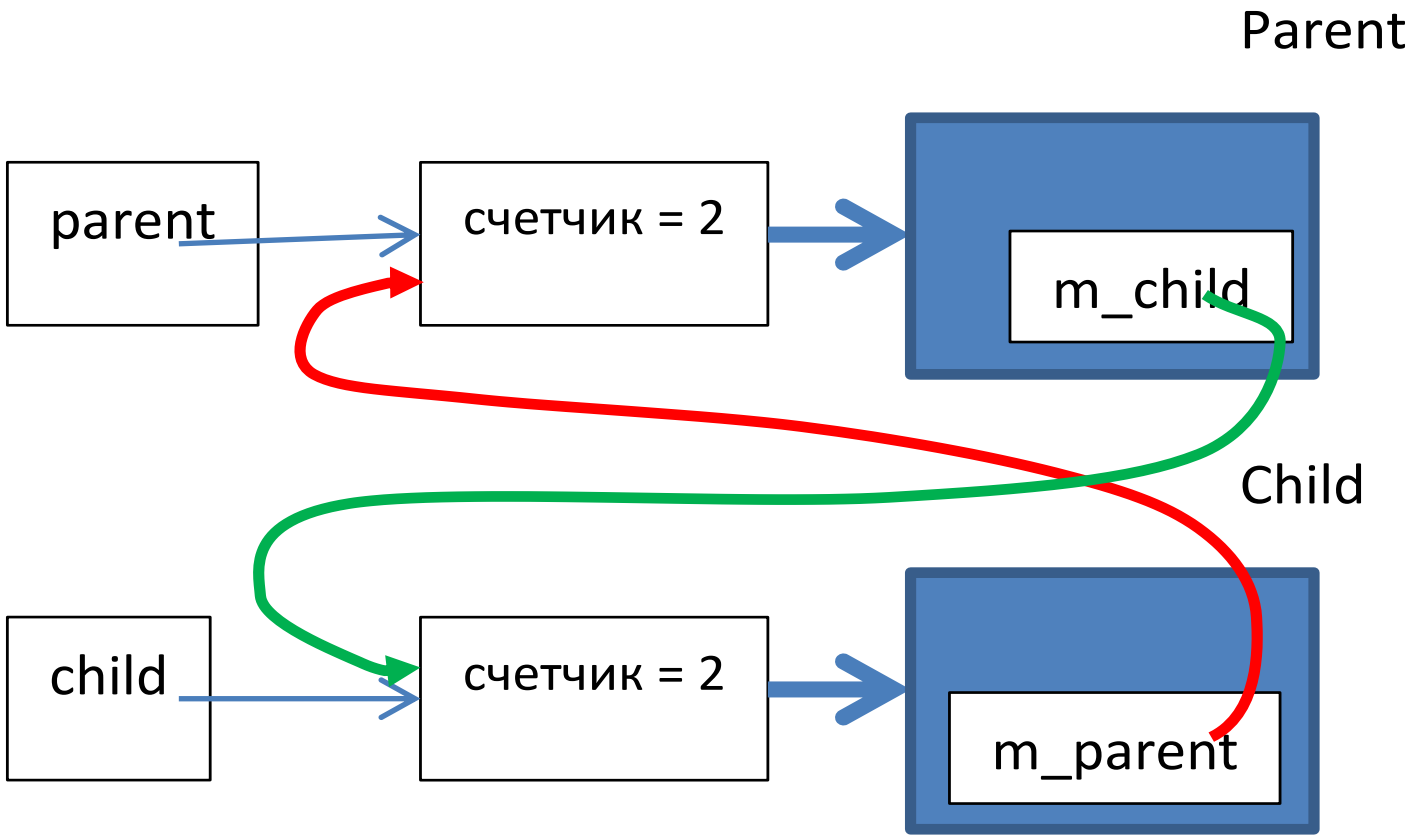
Пример циклических зависимостей

```
struct Child;
struct Parent {
        std::shared_ptr<Child> m_child;
};
struct Child {
        std::shared_ptr<Parent> m_parent;
};
```

Проблема циклических зависимостей

```
{
   std::shared_ptr<Parent> parent(new Parent());
   std::shared_ptr<Child> child(new Child());
   parent->m_child = child; //???
   child->m_parent = parent; //???
   int parentCount = parent.use_count(); //2
   int childCount = child.use_count();//2
}// ~child, ~parent ??? delete??? деструкторы???
```

Иллюстрация проблемы



В частности для решения таких проблем

используется std::**weak_ptr**!

STD::WEAK_PTR

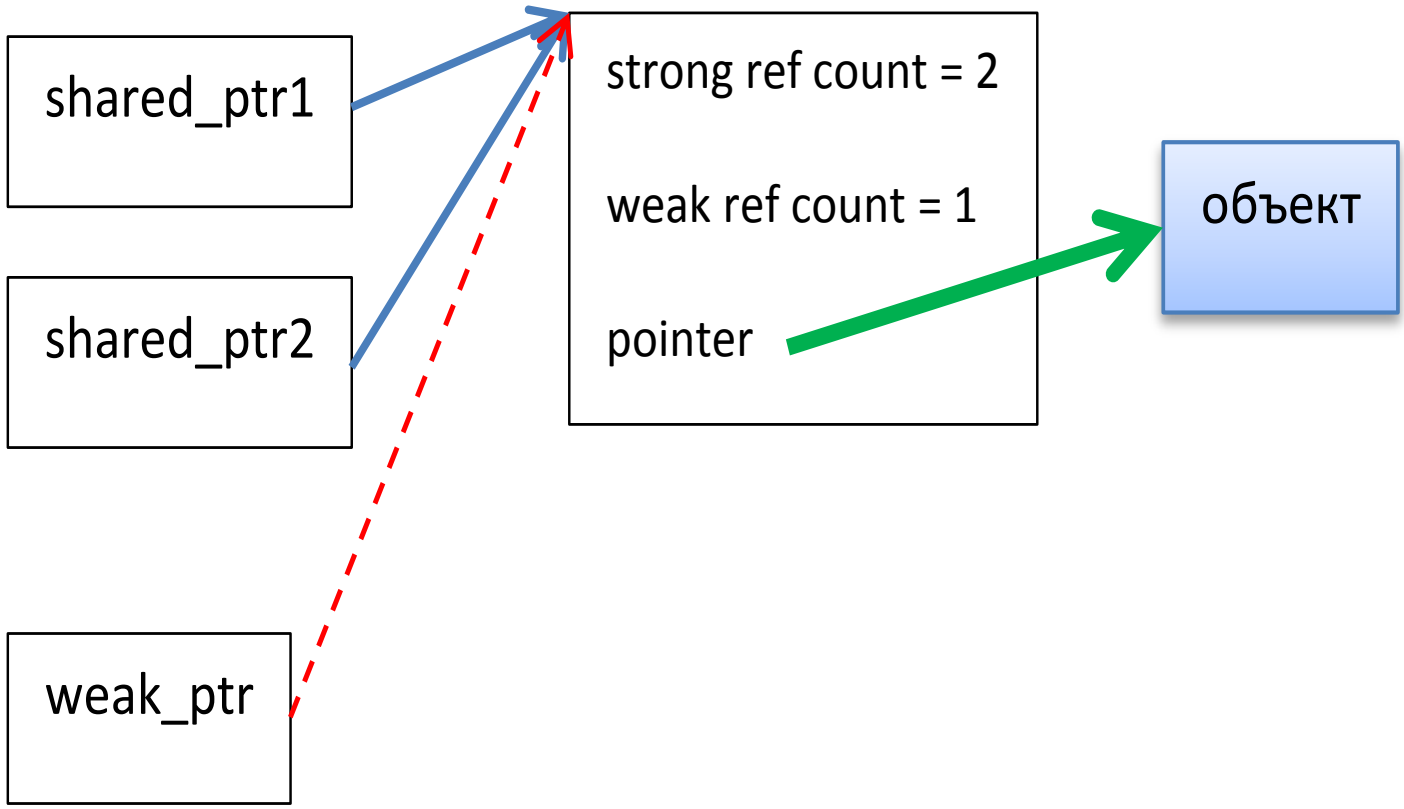
std::weak_ptr

- «слабый» указатель на объект, находящийся во владении shared_ptr - ов

- не учитывается при подсчете «сильных» ссылок (отдельный счетчик)
- не предоставляет **прямого** доступа к объекту
- не влияет на уничтожение целевого объекта
- при необходимости преобразуются в «сильный» (создается сильный и сч++), то есть shared_ptr посредством:
  - конструктора
  - operator=
  - метода **lock**()
    Если преобразование невозможно (пустой weak_ptr), генерируется исключение

Напоминание: weak_ptr используется, в основном, для разрешения циклических ссылок (в общем случае для временного использования)

Иллюстрация weak_ptr



Создание слабых указателей посредством сильных указателей

- Конструктор weak_ptr:

```
shared_ptr<A> sh(new A(1)); //strong count??? weak count???
weak_ptr<A> w(sh); //strong count??? weak count???
```

Создание слабых указателей посредством сильных указателей

- operator= :

weak_ptr<A> w; //”пустой”

shared_ptr<A> sh = make_shared<A>(1); //strong count??? weak count???

w=sh; //strong count??? weak count???

Использование целевого объекта посредством weak_ptr – конструктор shared_ptr

```
void f(weak_ptr<A>& wp)
{
   try {
        shared_ptr<A> sh(wp);
        sh->…
   }
catch (std::bad_weak_ptr& e) { return;}
}
```

Использование целевого объекта посредством weak_ptr – метод lock()

std::shared_ptr<T> **lock**() const

- если целевого объекта уже нет, то будет возвращен нулевой shared_ptr
- если объект «живой», то будет создан обычный shared_ptr на него (т.е. увеличен счетчик и т.д.) => гарантия существования

Использование целевого объекта посредством weak_ptr – метод lock()

```
void f(weak_ptr<A>& wp)
{
   if (  shared_ptr<A> sh = wp.lock() )
{
        sh->…
   }
}
```

bool weak_ptr::**expired**() const;

Проверка существования целевого объекта:

```
auto shp = std::make_shared<std::string>("abc");
std::weak_ptr<std::string> wptr = shp;
bool b1 = wptr.expired(); //???
shp.reset();
bool b2 = wptr.expired(); //???
```

Пример shared_ptr и weak_ptr

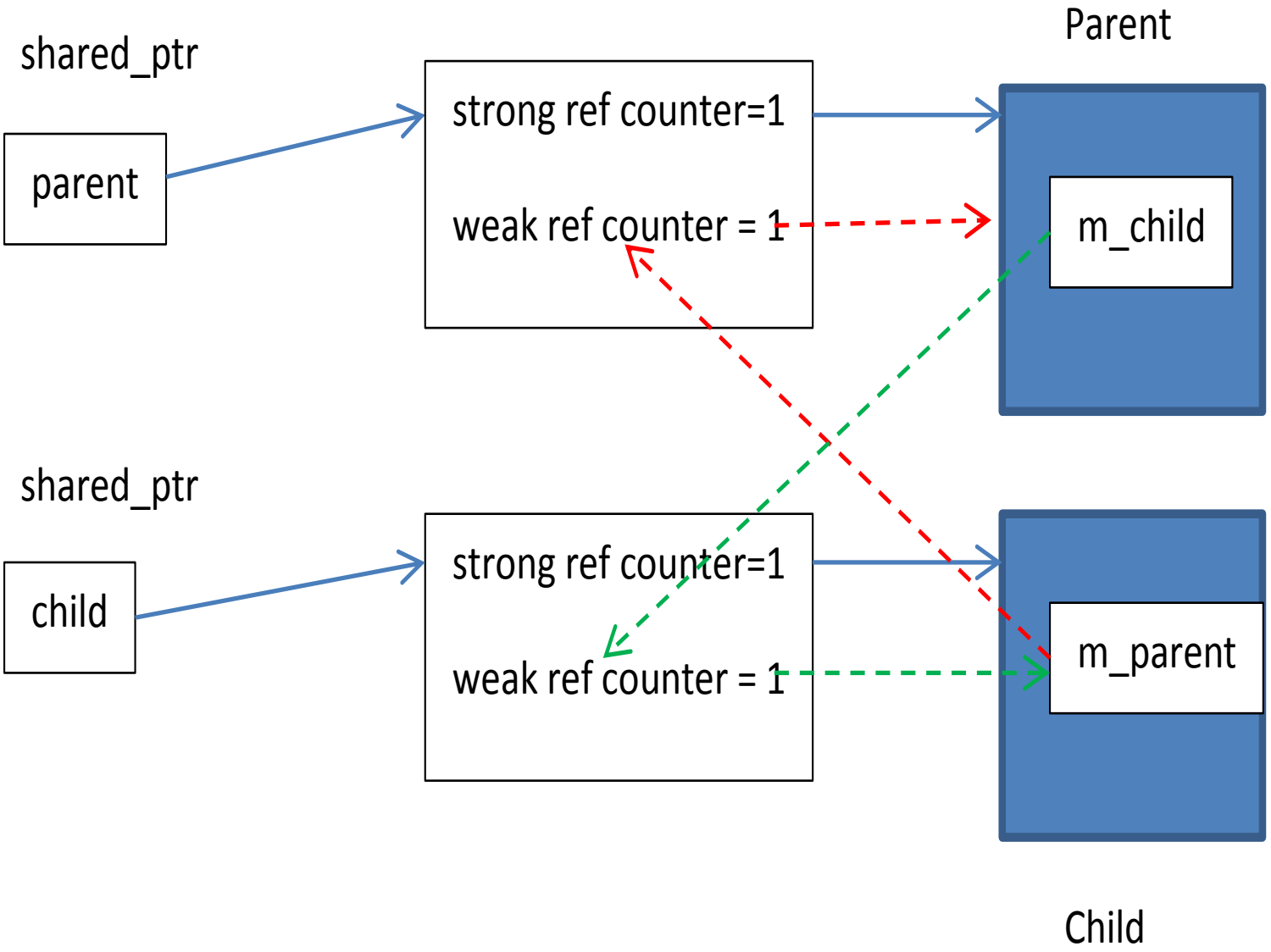
```
shared_ptr<A> a1( new A() ); //strong ref count=???, weak ref count=???
shared_ptr<A> a2 = a1; //??? ???
weak_ptr<A> w = a2; //??? ???

a2.reset(); //??? ???
// a2 - указатель ==nullptr,  но объект А жив и доступен через a1

   // так как на данный момент существует один сильный указатель
   // на объект (a1), то мы можем из слабого указателя получить сильный
a2 = w.lock();  //??? ???

a2.reset(); // ??? ???
a1.reset(); // ??? ??? => Объект ???

   // и попытка получить сильный указатель из слабого вернет пустой
shared_ptr<A> a3 = w.lock();
if(a3.get() == nullptr)  {  } //???
```

Модифицируем пример с циклическими ссылками

```
struct Child;
struct Parent {
        std::weak_ptr<Child> m_child;
};
struct Child {
        std::weak_ptr<Parent> m_parent;
};
```

Иллюстрация решения проблемы циклических ссылок

```
{
  std::shared_ptr<Parent> parent(new Parent());
  std::shared_ptr<Child> child(new Child());
  parent->m_child = child;
  child->m_parent = parent;
  int parentCount = parent.use_count(); //1
  int childCount = child.use_count(); //1
}
```


Parent
shared_ptr
parent
strong ref counter=1
weak ref counter = 1
m_child
shared_ptr
child
strong ref counter=1
weak ref counter = 1
m_parent
Child

Пример

```
std::weak_ptr<std::string> wptr ;
{
   auto shp1 = std::make_shared<std::string>("abc");
   std::cout<<shp1.use_count();//???
   wptr = shp1;
   std::cout<<shp1.use_count();//???
   auto shp2 = wptr.lock();//
   std::cout<<shp1.use_count();//???
}
bool b = wptr.expired(); //???
```

ENABLE_SHARED_FROM_THIS

Проблема: требуется создать shared_ptr в методе класса для this:

```
struct Bad {
   shared_ptr<Bad> getPtr()
                  {return shared_ptr<Bad>(this); } //???
};
int main() {
    std::shared_ptr<Bad> bp1(new Bad);
    std::shared_ptr<Bad> bp2 = bp1->getPtr();
    int count1 = bp1.use_count(); //1
    int count2 = bp2.use_count(); //1
} // ???
```

Решение:

```
struct Good: public std::enable_shared_from_this<Good>
{
   shared_ptr< Good > getPtr()
                    {return shared_from_this(); }
};
int main() {
    std::shared_ptr<Good> gp1(new Good);
    std::shared_ptr<Good> gp2 = gp1->getPtr();
    int count1 = gp1.use_count(); //2
    int count2 = gp2.use_count(); //2
} // ???
```

Реализация (обычная)

```
template<typename T>
class enable_shared_from_this {
    weak_ptr<T> weak_p;
public:
    shared_ptr<T> shared_from_this() {
        // Преобразование слабой ссылки в сильную посредством конструктора shared_ptr
        return shared_ptr<T>( weak_p );
        }
};
```

STATIC_ASSERT

static_assert

- осуществляет проверки во время компиляции
- и выдает заданную программистом диагностику

синтаксис:

static_assert(выражение_вычисляемое_на_этапе_компиляции, строка_диагностики);

Пример

```
template <typename T, size_t Size>
class MyArray {
        static_assert(Size < 10, "Size is too large");
        T m_ar[Size];
};

int main()
{
  MyArray<int, 26> a1; //error C2338: Size is too large
  return 0;
}
```

Пример

```
static_assert(
        __cplusplus > 199711L,
"Program requires C++11 capable compiler "
);
```

Шаблоны структур для проверок - что «предоставляет» класс

Можно использовать на этапе компиляции

#include <type_traits>

- is_**copy_constructible**, std::is_trivially_**copy_constructible**, is_nothrow_**copy_constructible**
- is_move_constructible<T>…
- …

Полезный пример

```
static_assert(std::is_copy_constructible<A>::value, "copy_constructible"); //OK

//static_assert(std::is_trivially_copy_constructible<A>::value, "trivially_copy_constructible"); //???

static_assert(std::is_copy_constructible<B>::value,
"copy_constructible");

static_assert(std::is_trivially_copy_constructible<B>::value, "trivially_copy_constructible");
```

Использование static_assert:

```
class A{
        ...
 A(const A&) = delete;
};

#include <type_traits>
template <typename T> void f( const T& a)
{
        static_assert(std::is_copy_constructible<T>::value,
                "No copy ctor");
        …
}
```

Exceptions

C++11

Новые возможности по обработке исключений

- язык C++
  - спецификация noexcept
  - оператор noexcept
- стандартная библиотека:
  - тип std::exception_ptr позволяет хранить исключение любого типа (стандарт не оговаривает реализацию => unspecified)
  - функции для работы с std::exception_ptr

Устаревшие спецификации функций

- Все предыдущие способы спецификаций функций – **deprecated** в C++11!
- => вместо:
  - void f() throw();
  - void f() throw(A, B);
  - void f() throw(…);

спецификация **noexcept** на все случаи жизни

Спецификация noexcept – альтернатива спецификации throw()

Способы использования:

- void f() noexcept {...} – не генерирует исключения
- void f() noexcept (выражение_вычисляемое на_этапе_компиляции) {...} - если true, то функция помечается как noexcept

оператор noexcept(<выражение>)

- выполняется на этапе компиляции
- возвращает true, если выражение не генерирует исключение

```
void f1() noexcept {…}
void f2(){…}
bool b1 =  noexcept(f1()) ; //true
bool b2 = noexcept(f2()) ; //false
```

std::exception_ptr

Специфика:

- typedef */*unspecified*/* exception_ptr;
- хранит указатель на объект-исключение
- по умолчанию == null pointer
- два экземпляра равны только в том случае, когда или оба содержат нулевые указатели, или указатели на один и тот же объект-исключение

std::exception_ptr

Для использования std::exception_ptr стандартная библиотека предоставляет функции <exception>:

- **current_exception**()
- **rethrow_exception()**
- **make_exception_ptr()**

std::exception_ptr std::current_exception()

- обычно вызывается в обработчике исключения, иначе возвращает «пустой» std::exception_ptr
- формирует :
  - копию объекта-исключения, или ссылку на уже существующий объект-исключение, если ОК
  - объект std::bad_alloc (если реализация функции использует new) или объект std::bad_exception

void std::rethrow_exception(std::exception_ptr)

“перебрасывает” исключение на уровень выше

Пример

```
void f(std::exception_ptr& e){
        try{
                if (e){std::rethrow_exception(e);}
}
catch (int n){…}
catch (const char* str){…}
}

int main(){
        std::exception_ptr ex;
        try{
        //throw 1;
        throw "error";
        }
        catch (...){ex = std::current_exception();}
        f(ex);
}
```

template<class E> std::exception_ptr **make_exception_ptr**(E e)

создает std::exception_ptr , который содержит копию e или ссылку на e