



ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ

ЛЕКЦИЯ №5

Exceptions

Для реализации механизма обработки исключений в язык Си++ введены следующие три ключевых (служебных) слова:

- 1.** **try** (контролировать)
- 2.** **catch** (ловить)
- 3.** **throw** (генерировать, порождать, бросать, посыпать, формировать).

Добавляем в квадратное уравнение проверку

Example32_FirstException

Куда добавить генерацию исключения?

```
if ((b*b-4*a*c)<0) throw WrongArgumentException();
```

1. Конструктор
2. Методы вычисления решения

```
SquareEquation se(a, b, c);  
try {  
    std::cout << "X1=" << se.FindX1() << "\n";  
    std::cout << "X2=" << se.FindX2() << "\n";  
} catch (WrongEquationException ex) {  
    std::cout << std::endl << ex.what() << std::endl;  
}
```



Exception

Исключение это:

1. Объект, наследник класса std::exception (#include <exception>)
2. Событие, прерывающее обработку программы.

Под прерыванием мы понимаем, что срабатывание исключение, аналогично срабатыванию оператора return.

Но есть существенное отличие: return возвращает результат в то место, где вызвали функцию. Исключение возвращает объект исключения только в те места, где его явно ловят (catch)! И только если исключение сработало (throw) в месте где мы его контролируем (try)!

Если исключение не поймать (catch) то оно будет прерывать работу функций, поднимаясь вверх по стеку вызова, пока не остановит программу.



Try / Catch

Служебное слово try позволяет выделить в любом месте исполняемого текста программы так называемый контролируемый блок:

```
try {  
    //операторы  
    //операторы  
} catch (Тип_исключения1 имя) {  
    //операторы  
} catch (Тип_исключения2 имя) {  
    //операторы  
}
```



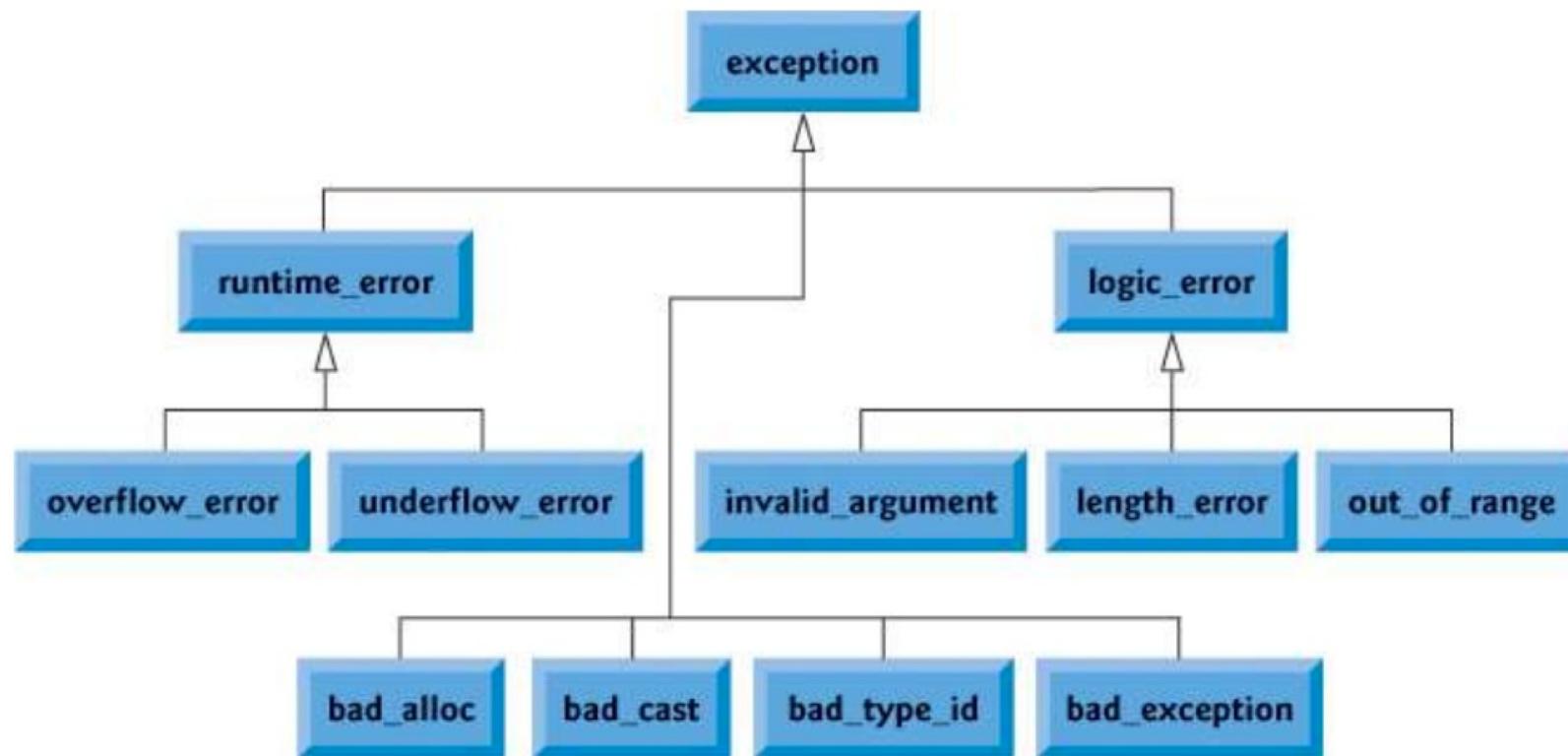
Exception – это все что угодно

C++ позволяет создавать исключения любого типа, хотя обычно рекомендуется создавать типы, производные от `std::exception`. Исключение в C++ может быть перехвачено обработчиком `catch`, в котором определен тот же тип, что и у созданного исключения, или обработчиком, который способен перехватывать любой тип исключения.

Если созданное исключение имеет тип класса, у которого имеется один или несколько базовых классов, то его могут перехватывать обработчики, которые принимают базовые классы (и ссылки на базовые классы) этого типа исключения. Обратите внимание, что если исключение перехватывается по ссылке, то оно привязывается к самому объекту исключения; в противном случае обрабатывается его копия (как и в случае с аргументами функции).



Стандартные исключения C++



Повторное «возбуждение» исключений

```
try
{
    -----
    try
    {
        -----
        throw val;
        -----
    } catch(data-type arg)
    {
        -----
        throw;
        -----
    }
    -----
    -----
}
catch(data-type arg)
{
    -----
    -----
}
```

Exceptions

Example34_ComplexException

```
try{

throw MyException(100);

}catch(MyException& ex) {

    std::cout << "MyException in Proc2: " << ex.A << "\n";

    std::exception_ptr currentException = std::make_exception_ptr(ex);

    std::rethrow_exception(currentException);

}

catch(...) {

    std::exception_ptr currentException = std::current_exception();

}
```



Созданное исключение может перехватываться следующими типами обработчиков `catch`:

1. Обработчик, который может принимать любой тип данных (синтаксис с многоточием).
2. Обработчик, который принимает тот же тип, что и у объекта исключения; поскольку используется копия объекта, то модификаторы `const` и `volatile` игнорируются.
3. Обработчик, который принимает ссылку на тот же тип, что и у объекта исключения.
4. Обработчик, который принимает ссылку на форму `const` или `volatile` того же типа, что и у объекта исключения.
5. Обработчик, который принимает базовый класс того же типа, что и у объекта исключения; поскольку используется копия объекта, то модификаторы `const` и `volatile` игнорируются. Обработчик `catch` для базового класса не должен предшествовать обработчику `catch` для производного класса.
6. Обработчик, который принимает ссылку на базовый класс того же типа, что и у объекта исключения.
7. Обработчик, который принимает ссылку на форму `const` или `volatile` базового класса того же типа, что и у объекта исключения.
8. Обработчик, который принимает указатель, в который можно преобразовать созданный объект указателя при помощи стандартных правил преобразования указателей.

Очистка стека

В механизме исключений C++ элемент управления перемещается из оператора `throw` в первый оператор `catch`, который может обработать выданный тип. При достижении оператора `catch` все автоматические переменные, находящиеся в области между операторами `throw` и `catch`, удаляются в процессе, который называется очистка стека. При очистке стека выполнение продолжается следующим образом.

1. Элемент управления достигает оператора `try` в процессе нормального последовательного выполнения. Выполняется защищенный раздел в блоке `try`.
2. Если во время выполнения защищенного раздела исключение не создается, предложения `catch`, следующие за блоком `try`, не выполняются. Выполнение продолжается с оператора, расположенного после последнего предложения `catch`, следующего за связанным блоком `try`.



Очистка стека (продолжение)

3. Если исключение создается во время выполнения защищенного раздела или во время выполнения любой подпрограммы, прямо или косвенно вызываемой защищенным разделом, объект исключения создается из объекта, созданного операндом **throw**. (Это означает, что может быть задействован конструктор копии.) Обработчики **catch** проверяются в порядке их отображения после блока **try**. Если соответствующий обработчик не найден, проверяется следующий динамический внешний блок **try**. Этот процесс будет продолжаться до тех пор, пока не будет проверен последний внешний блок **try**.
4. Если соответствующий обработчик **catch** найден и он выполняет перехват по значению, его формальный параметр инициализируется путем копирования объекта исключения. Если обработчик выполняет перехват по ссылке, параметр инициализируется для ссылки на объект исключения. После инициализации формального параметра начинается процесс очистки стека. Выполняется обработчик **catch**, и выполнение программы продолжается после последнего обработчика, то есть с первого оператора или конструкции, которые не являются обработчиком **catch**.

Exception

Example35_ExceptionCatch

Помни, что в блоке catch могут возникать свои исключения!

Если в блоке catch идет высвобождение ресурсов (удаление объектов, закрытие дескрипторов файлов) то их надо самих помещать в еще один вложенный блок try/catch.



Обработка всех исключений

```
1. try {  
2.     throw CSomeOtherException();  
3. }  
4. catch(...) {  
5.     // Catch all exceptions - dangerous!!!  
6.     // Respond (perhaps only partially) to the exception, then  
7.     // re-throw to pass the exception to some other handler  
8.     // ...  
9.     throw;  
10. }
```



noexcept

Example35_noexcept

В стандарте ISO C++11 был представлен оператор **noexcept**. Он поддерживается в Visual Studio 2015 и более поздних версий. Когда это возможно, используйте noexcept, чтобы задать возможность вызова функцией исключений.

В предыдущих версиях стандарта можно использовать **throw();**

Если метод с noexcept все таки сгенерирует исключение, то оно перехвачено уже не будет.



Если исключение не обработали

Example36_terminate

Если для текущего исключения не удается найти подходящий обработчик (или обработчик `catch` для многоточия), то вызывается предопределенная функция времени выполнения `terminate`. (Функцию `terminate` также можно явным образом вызвать из любого обработчика.)

Действие по умолчанию для `terminate` заключается в том, что она вызывает функцию `abort`. Если вам необходимо, чтобы перед выходом из приложения функция `terminate` в вашей программе вызывала какую-то другую функцию, вызовите функцию `set_terminate`, указав в качестве ее единственного аргумента ту функцию, которую нужно вызвать.

Функцию `set_terminate` можно вызвать из любого места программы. Процедура `terminate` всегда вызывает последнюю функцию, заданную в качестве аргумента для `set_terminate`.



Exceptions

ИТОГО

1. Помогает создать надежную программу;
2. Отделяет код обработки ошибок от основной логики программы;
3. Обработка исключений может быть реализована за пределами основного кода программы;
4. Существует возможность обрабатывать только выбранные типы исключений;
5. Программа, обрабатывающая исключения не остановится без объяснения причин (например, вывода на экран причины возникновения исключений);



Умные указатели

ЛЕКЦИЯ №6



RAII

Resource Acquisition Is Initialization

Получение ресурса есть инициализация (RAII) — программная идиома объектно-ориентированного программирования, смысл которой заключается в том, что с помощью тех или иных программных механизмов получение некоторого ресурса неразрывно совмещается с инициализацией, а освобождение — с уничтожением объекта.

Типичным (хотя и не единственным) способом реализации является организация получения доступа к ресурсу в **конструкторе**, а освобождения — в **деструкторе** соответствующего класса.

Поскольку деструктор автоматической переменной вызывается при выходе её из области видимости, то ресурс гарантированно освобождается при уничтожении переменной. Это справедливо и в ситуациях, в которых возникают **исключения**.



Пара слов о шаблонах (template)

подробнее на следующей лекции

Шаблоны (англ. template) — средство языка C++, предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов, значениям по умолчанию).

Шаблоны позволяют создавать **параметризованные классы и функции**. Параметром может быть любой тип или значение одного из допустимых типов (целое число, enum, указатель на любой объект с глобально доступным именем, ссылка)

Использование шаблонов классов:

имя_шаблона<параметр_шаблона_1,параметр_шаблона2> имя_объекта



Шаблон std::shared_ptr<T>

```
#include<memory>
```

1. Предоставляет возможности по обеспечению автоматического удаления объекта, за счет подсчета ссылок указатели на объект;
2. Хранит ссылку на один объект;
3. При создании std::shared_ptr<T> счетчик ссылок на объект увеличивается;
4. При удалении std::shared_ptr<T> счетчик ссылок на объект уменьшается;
5. При достижении счетчиком значения 0 – объект автоматически удаляется;



SharedPtr

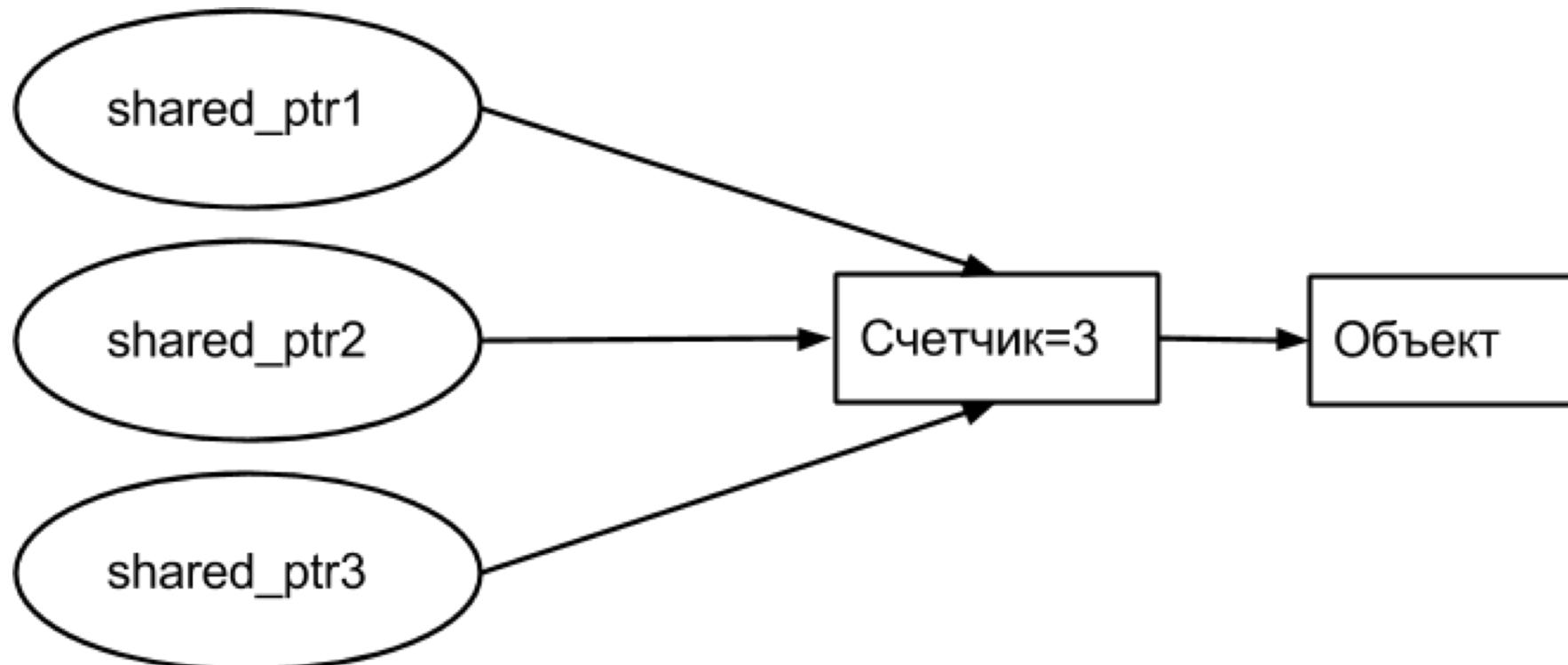
Example38_SharedPtr

```
1. void Foo(std::shared_ptr<A> a) { //shared_ptr передаем по значению, сам объект не копируется
2.     std::shared_ptr<A> b(a); // Копируем ссылку
3.     a->DoSomething(); // C shared_ptr мы можем работать как с обычным указателем
4.     b->DoSomething();
5. }

6. int main(int argc, char** argv) {
7.     std::shared_ptr<A> a(new A()); //В передаем указатель на нужный объект
8.     Foo(a);
9.     return 0;
10.}
```



Простой подсчет ссылок на объекты (то есть, копий shared_ptr)



Более реалистичный пример

Example39_SharedPtr2

```
1. class A {
2. private:
3.     std::shared_ptr<A> next;
4. public:
5.     A() {
6.         std::cout << "I'm alive!" << std::endl;
7.     }
8.     A(A* next_ptr) : A(){
9.         next = std::shared_ptr<A>(next_ptr);
10.    }
11.    virtual ~A() {
12.        std::cout << "O no! I'am dead!" << std::endl;
13.    }
14.};
15.int main(int argc, char** argv) {
16.    std::shared_ptr<A> a(new A(new A(new A(new A())))); // Ни один объект не потерянся
17.    return 0;}
```



Наследование работает как обычно

Example45_SharedPtrInheretance

```
1. class A {  
2. private:  
3.     const char* name;  
4. public:  
5.     A(const char*value) : name(value) {};  
6. };  
7.  
8. class B : public A {  
9. public:  
10.    B(const char*value) : A(value) {};  
11. };  
  
12. int main(int argc, char** argv) {  
13.     std::shared_ptr<A> b(new B("My name is B!"));  
14.     b->Print();}
```



Swap – обмениваемся указателями

Example40_SharedPtr3

```
1.int main(int argc, char** argv) {  
2.    std::shared_ptr<A> a(new A("My name is A"));  
3.    std::shared_ptr<A> b(new A("My name is B"));  
4.    std::swap(a,b);  
  
5.//template <class T>  
6.//void swap (shared_ptr<T>& x, shared_ptr<T>& y) noexcept;  
  
7.    a->WhoAmI();  
8.    b->WhoAmI();  
9.    return 0;  
10.}
```



Внешний деструктор

Example41_SharedPtr4

```
1. void deleter(A *a){  
2.     std::cout << "Nobody kills in my ship!" << std::endl;  
3. }  
  
4. int main(int argc, char** argv) {  
5.     A a("My name is A");  
6.     std::shared_ptr<A> a_ptr(&a,&deleter); // Вместо деструктора вызывается наша функция  
7.     return 0;  
8. }
```

shared_ptr в списке параметров функций

Example48_SharedPtr5

```
1.try{  
2.    /* Так плохо!  
3.    A *a=new A("A");  
4.    foo(foofoofoo( ),std::shared_ptr<A>(a));  
5.    /*  
6.    std::shared_ptr<A> a(new A("B" ));  
7.    foo(foofoofoo( ),a);  
8.    /*  
9.    }catch(...){  
10. }
```



std::dynamic_pointer_cast<T>

example42_dynamicpointercast

```
std::shared_ptr<B> b(new B());  
  
std::shared_ptr<C> c(new C());  
std::shared_ptr<A> ptr = b;  
  
if(std::shared_ptr<B> ptr_b = std::dynamic_pointer_cast<B>(ptr) ) {  
    ptr_b->Do();  
}  
  
if(std::shared_ptr<C> ptr_c = std::dynamic_pointer_cast<C>(ptr) ) {  
    ptr_c->Do();  
}
```



std::make_shared<T>

example42_makeshared

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

int main () {

    std::shared_ptr<int> foo = std::make_shared<int> (10);
    // same as:

    std::shared_ptr<int> foo2 (new int(10));
    std::cout << "*foo: " << *foo << '\n';
    std::cout << "*foo2: " << *foo2 << '\n';

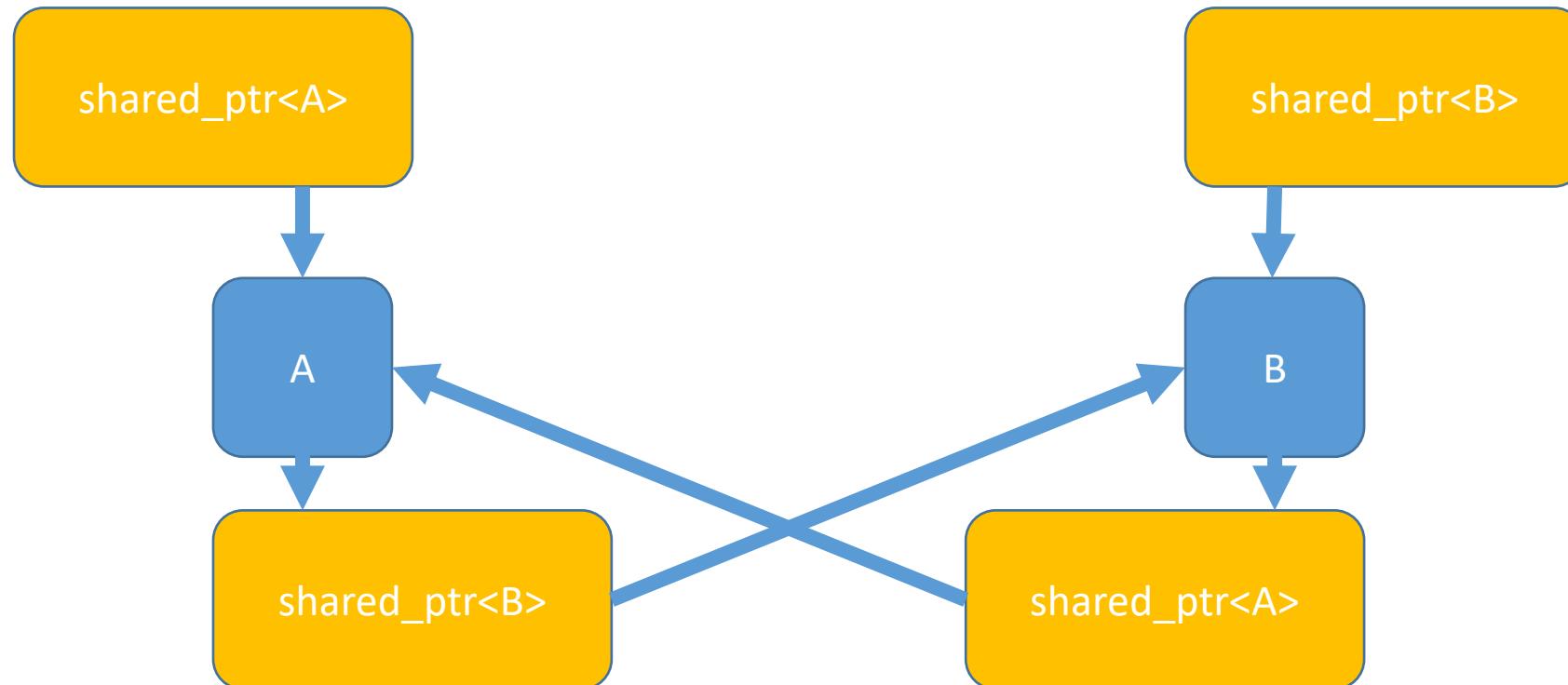
    return 0;
}
```



Перекрестные ссылки и std::shared_ptr

Example42_SharedPtrDeadlock

Если зациклить объекты друг на друга, то появится «цикл» и объект ни когда не удалится! Т.к. деструктор не запустится!



Слабый указатель std::weak_ptr

shared_ptr представляет *разделяемое владение*, но с моей точки зрения разделяемое владение не является идеальным вариантом: значительно лучше, когда у объекта есть конкретный владелец и его время жизни точно определено.

1. Обеспечивает доступ к объекту, только когда он существует;
2. Может быть удален кем-то другим;
3. Содержит деструктор, вызываемый после его последнего использования (обычно для удаления анонимного участка памяти).



Теперь без dead lock

Example43_Weak_Ptr

```
1.class A {  
2.private:  
3.    std::weak_ptr<B> b;  
4.public:  
5.    void LetsLock(std::shared_ptr<B> value) {  
6.        b = value;  
7.    }  
8.    ~A( ){  
9.        std::cout << "A killed!" << std::endl;  
10.    }  
11.};
```



Объекты, не являющиеся владельцем

Example44_Weak_Ptr2

```
1.class WeakPrint {
2.private:
3.    std::weak_ptr<A> array[5];
4.public:
5.    WeakPrint(std::shared_ptr<A> value[5]) {
6.        for (int i = 0; i < 5; i++)
7.            array[i] = value[i];
8.    }
9.    void Print() {
10.        for (int i = 0; i < 5; i++)
11.            if (std::shared_ptr<A> a = array[i].lock())
12.                a->Print();
13.    }
14.};
```



unique_ptr<T>

unique_ptr (определен в **<memory>**) и обеспечивает семантику строгого владения.

- Владеет объектом, на который хранит указатель.
- Не CopyConstructable и не CopyAssignable, однако MoveConstructible и MoveAssignable.
- При собственном удалении (например, при выходе из области видимости (6.7)) уничтожает объект (на который хранит указатель) с помощью заданного метода удаления (с помощью deleter-a).

Использование **unique_ptr** дает:

- безопасность исключений при работе с динамически выделенной памятью,
- передачу владения динамически выделенной памяти в функцию,
- возвращения динамически выделенной памяти из функции,
- хранение указателей в контейнерах



Исключительное право владения

Example46_UniquePtr

```
1. A* unsafe_function( ){
2.     A* a = new A("I', only one!");
3.     return a;      }
4. std::unique_ptr<A> safe_function( ){
5.     std::unique_ptr<A> a(new A("I', only one!"));
6.     return a;
7. }
```

Почему unique_ptr не копируется?

Example47_UiquePtr2

Единственные доступные операторы копирования:

```
unique_ptr& operator= (unique_ptr&& x) noexcept; // копируем из rvalue  
assign null pointer (2) unique_ptr& operator= (nullptr_t) noexcept
```

Для явного перемещения содержимого можно использовать std::move



std::move

```
template< class T >
typename std::remove_reference<T>::type&& move( T&& t );
```

Возвращает объект LValue с помощью шаблона структуры `std::remove_reference`, которая помогает получить тип без ссылок:

```
template< class T > struct remove_reference {typedef T type;};
template< class T > struct remove_reference<T&> {typedef T type;};
template< class T > struct remove_reference<T&&> {typedef T type};
```





Спасибо!

ВСЕ ИДЕМ НА ПЕРЕРЫВ