Modern C++

Move Semantic. Smart Pointers.

Евгений Козлов

12.11.2018

CQG

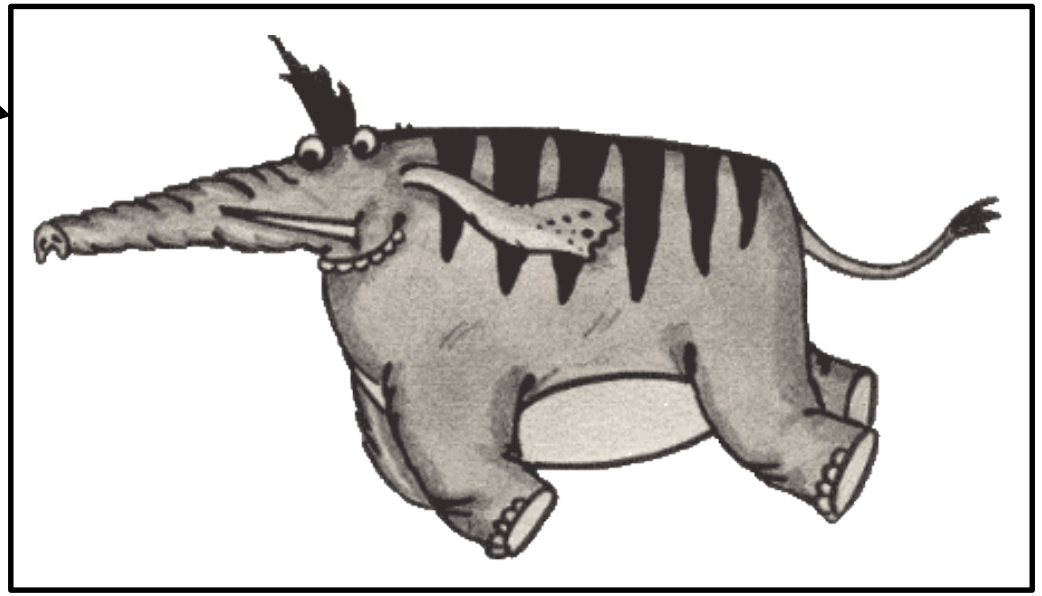
Семантика перемещения

CQG

Что такое vector?

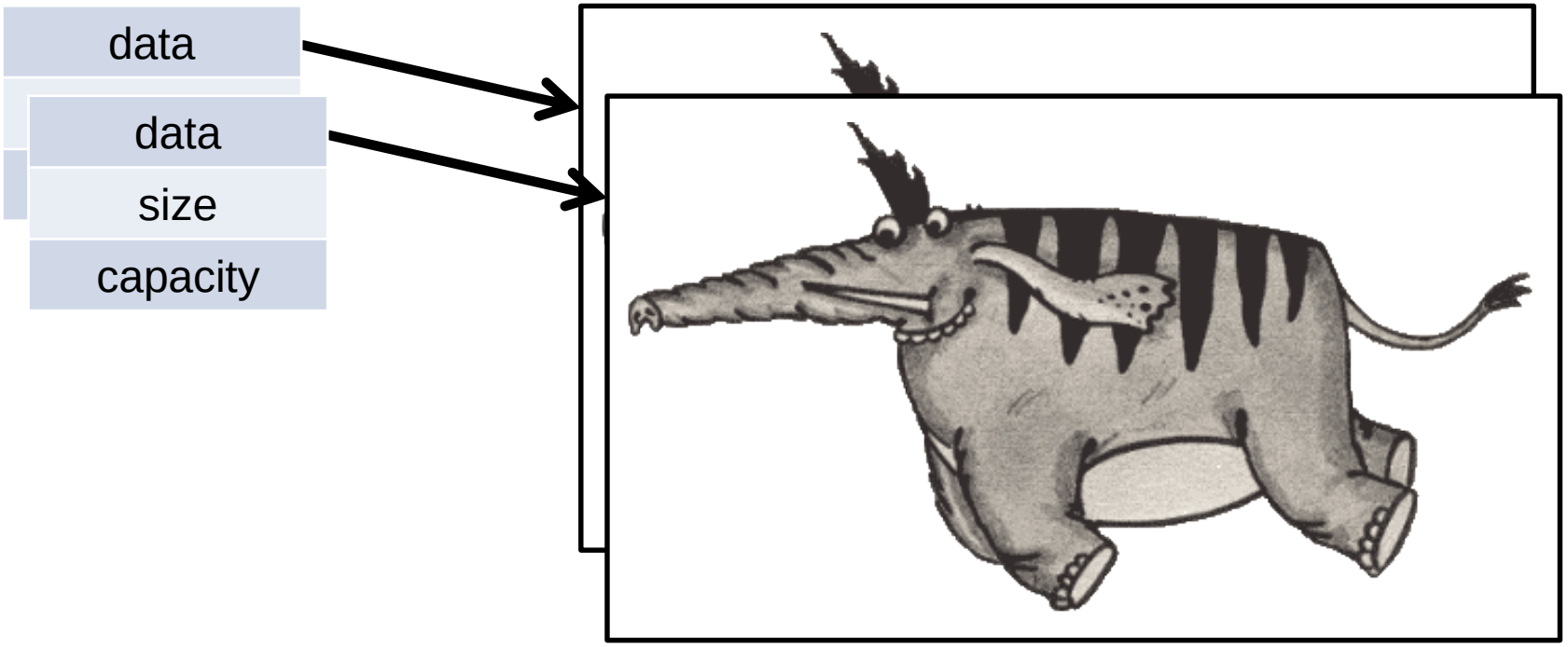
data
size
capacity

12 или 24 байта



??? байт

Что такое vector?



Копирование больших объектов

```
void PrintLine(string i_text)
{
    ...
}

int main()
{
    string text = ...;
    PrintLine(text);
    ...
}
```

Копирование больших объектов

```
void PrintLine(const string& i_text)
{
    ...
}

int main()
{
    string text = ...;
    PrintLine(text);
    ...
}
```

Копирование больших объектов

```
string ReadLine()
{
    string text = ...;
    return text;
}

int main()
{
    string textCopy = ReadLine();
    ...
}
```

Копирование больших объектов

```
void ReadLine(string& o_text)
{
    o_text = ...;
}

int main()
{
    string text;
    ReadLine(text);
    ...
}
```

Копирование больших объектов

```
void Object::SetText(const string& text)
{
    m_text = text;
}

void Object::UpdateText()
{
    string text = ReadLine();
    if (IsValid(text))
        SetText(text);
}
```

Копирование больших объектов

```
void Object::SetText(string& text)
{
    swap(m_text, text);
}

void Object::UpdateText()
{
    string text = ReadLine();
    if (IsValid(text))
        SetText(text);
}
```

CQG

Копирование больших объектов

Проблема 1: очень легко использовать неправильно

```
void Object::SetText(string& text) { ... }
void Object::UpdateText()
{
    string text = ReadLine();
    if (IsValid(text))
    {
        SetText(text);
        PrintLine("Text updated: " + text); // Баг
    }
}
```

Копирование больших объектов

Проблема 2: нельзя использовать с временными объектами

```
void Object::SetText(string& text) { ... }
void Object::UpdateText()
{
    string text = ReadLine();
    if (IsValid(text))
        SetText(text);
    else
        SetText("Invalid text"); // Ошибка компиляции
}
```

CQG

R-value reference

1) `T&`
2) `const T&`
3) `T&&`

R-value reference – новый тип ссылок:

- **«Ссылка на временный объект»**
- **Гарантируется, что объект, лежащий по этой ссылке, нигде больше не используется**
- **Можно получить из любого неконстантного объекта с помощью вызова `std::move`**

CQG

R-value reference

Нельзя использовать «случайно»

```
void Object::SetText(string&& text) { ... }
void Object::UpdateText()
{
    string text = ReadLine();
    if (IsValid(text))
    {
        SetText(text); // Ошибка компиляции
        PrintLine("Text updated: " + text);
    }
}
```

CQG

R-value reference

Нельзя использовать «случайно»

```
void Object::SetText(string&& text) { ... }
void Object::UpdateText()
{
    string text = ReadLine();
    if (IsValid(text))
    {
        SetText(std::move(text));
        PrintLine("Text updated!");
    }
}
```

CQG

R-value reference

Можно использовать с временными объектами

```
void Object::SetText(string&& text) { ... }
void Object::UpdateText()
{
    string text = ReadLine();
    if (IsValid(text))
        SetText(std::move(text));
    else
        SetText("Invalid text"); // OK
}
```

Семантика перемещения

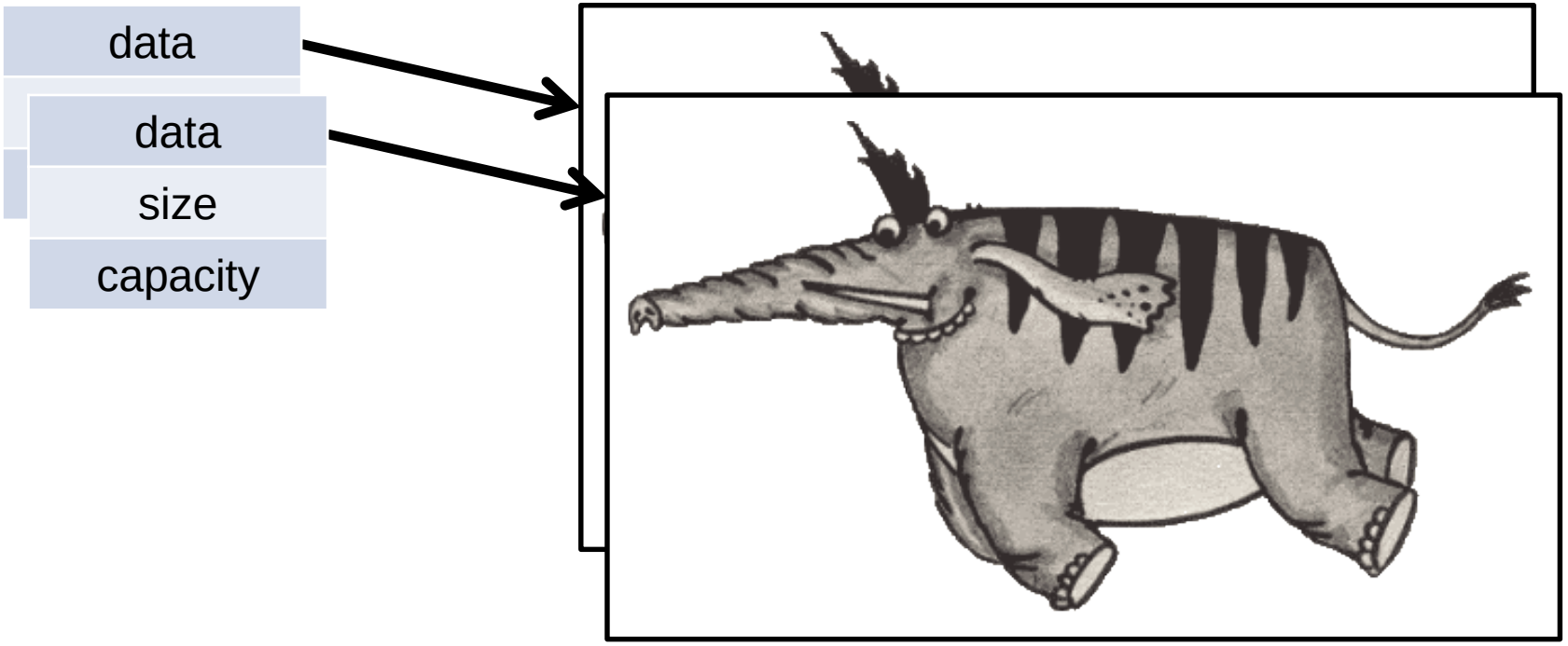
```
class A
{
public:
    A(); // default ctor
    A(int); // ctor with parameters

    A(const A&); // copy ctor
    A& operator=(const A&); // assignment op

    A(A&&); // move ctor
    A& operator=(A&&); // move-assignment op
};
```

Семантика перемещения

```
void Object::SetText(string&& text)
{
    // swap(m_text, text)
    m_text = text;
}
```

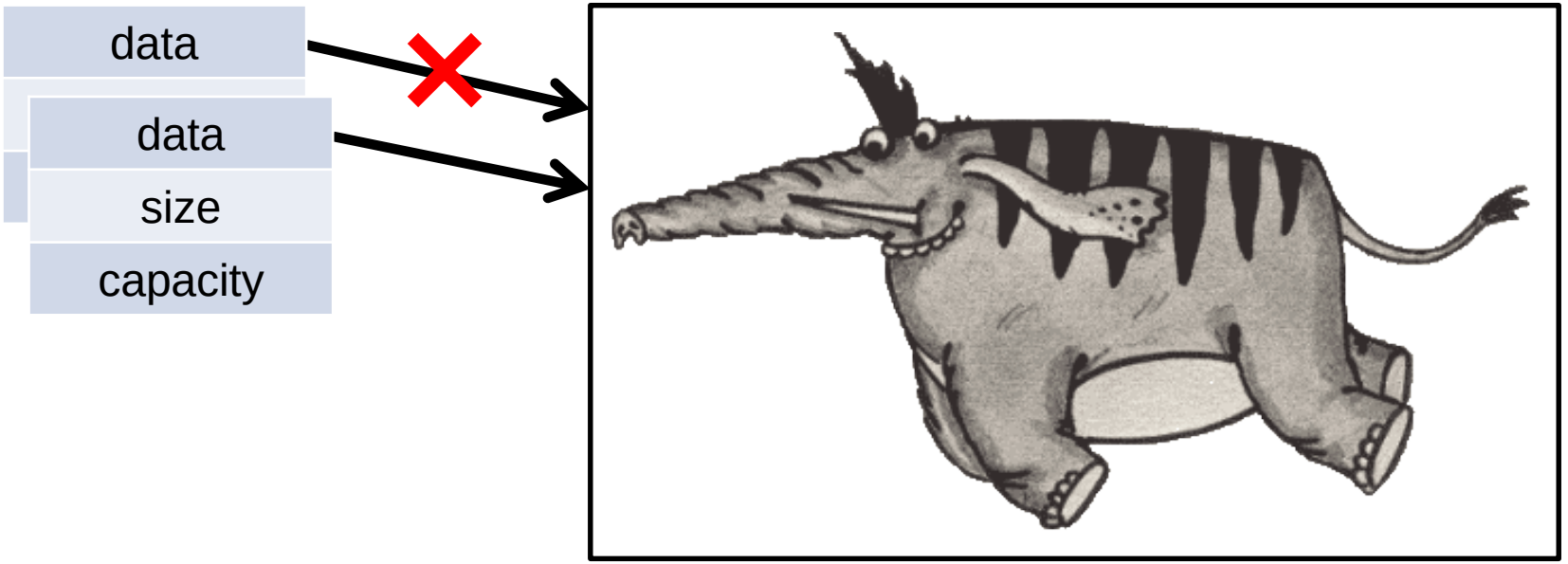
Семантика перемещения

Копирование:



Семантика перемещения

Перемещение:



Семантика перемещения

Необходимые и достаточные условия, при которых произойдет перемещение:

- **Компилятор знает как перемещать объект данного типа (определен move-коструктор, move-оператор присваивания).**

- **Компилятор знает, что в этом месте разрешено переместить объект (в правой части выражения находится R-value reference).**

Семантика перемещения

Move-конструктор и move-оператор присваивания будут сгенерированы автоматически, если:

- **Нет деструктора**
- **Нет copy-конструктора/оператора присваивания**
- **Нет move-конструктора/оператора присваивания**

CQG

Пример

```
class IntArray
{
    size_t m_size;
    int* m_data;

public:
    IntArray(int i_size)
        : m_size(i_size)
        , m_data(new int[i_size]) {}

    ~IntArray() { delete[] m_data; }
    ...
```

Copy-конструктор

```
...
IntArray(const IntArray& i_other)
    : m_size(i_other.m_size)
    , m_data(new int[m_size])
{
    memcpy(m_data, i_other.m_data, m_size);
}
...
```

Оператор присваивания

```
...
IntArray& operator=(const IntArray& i_other)
{
   if (this == &i_other) return;
   delete[] m_data;
   m_size = i_other.m_size;
   m_data = new int[m_size];
   memcpy(m_data, i_other.m_data, m_size);
   return *this;
}
...
```

Move-конструктор

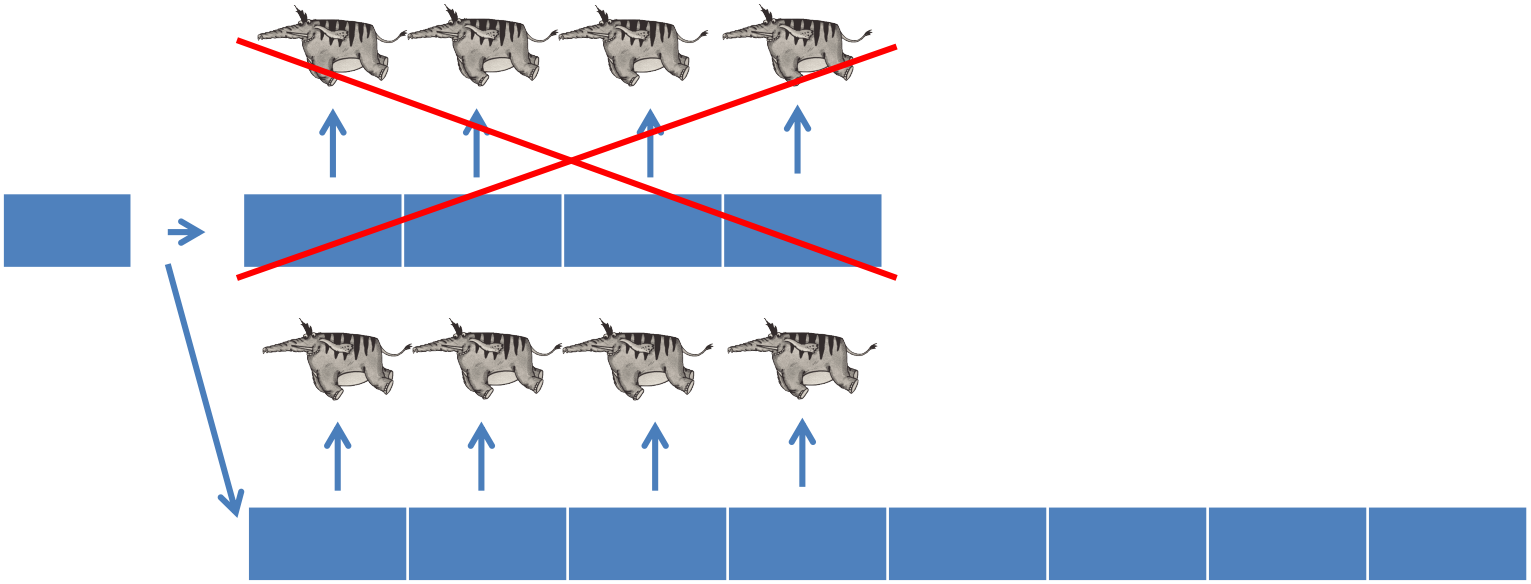
```
...
IntArray(IntArray&& i_other)
    : m_size(i_other.m_size)
    , m_data(i_other.m_data)
{
    i_other.m_size = 0;
    i_other.m_data = nullptr;
}
...
```

Move-оператор присваивания

```
...
IntArray& operator=(IntArray&& i_other)
{
    delete[] m_data;

    m_size = i_other.m_size;
    m_data = i_other.m_data;

    i_other.m_size = 0;
    i_other.m_data = nullptr;
    return *this;
}
```

vector of vectors

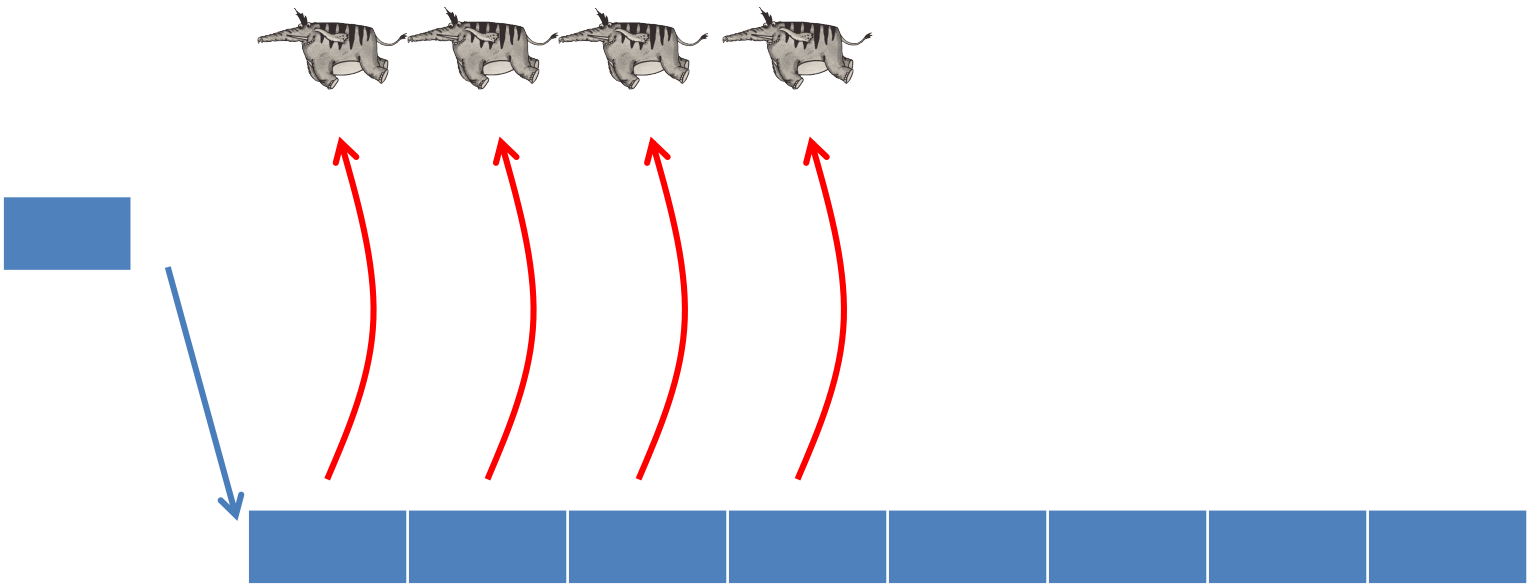
```
vector<vector<int>> collection;
collection.push_back(...);
```



CQG

vector of vectors

```
vector<vector<int>> collection;
collection.push_back(...);
```



CQG

noexcept

```
class A
{
public:
    A(); // default ctor
    A(int); // ctor with parameters

    A(const A&); // copy ctor
    A& operator=(const A&); //assignment op

    A(A&&) noexcept; // move ctor
    A& operator=(A&&) noexcept; // move-assignment op
};
```

Умные указатели

CQG

Работа с указателями

```
void ExampleMethod()
{
    int* pt(new int);
    ...
    delete pt;
}
```

int* – raw pointer

Работа с указателями

```
bool f(...)
{
    int* pt(new int);
    ...
    if (...)
    {
        delete pt;
        return false;
    }
    ...
    delete pt;
    return true;
}
```

Работа с указателями

```
bool f(...)
{
    int* pt(new int);
    ...
    if (...)
    {
        delete pt;
        throw CException();
    }
    ...
    delete pt;
    return true;
}
```

Работа с указателями

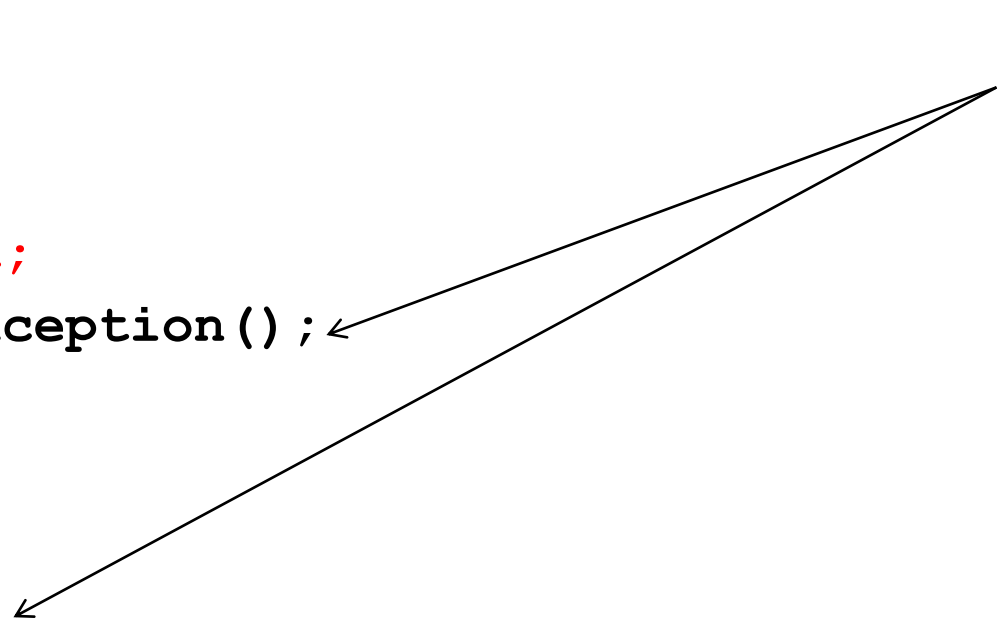
```
bool f(...)
{
   int* pt(new int);
   ...
   if (...)
   {
      SomethingThatCanThrow(pt); // ?

   }
   ...
   delete pt;
   return true;
}
```

Работа с указателями

```
bool f(...)
{
    int* pt(new int);
    ...
    if (...)
    {
        try { SomethingThatCanThrow(pt); }
        catch (...) { delete pt; throw; }
    }
    ...
    delete pt;
    return true;
}
```

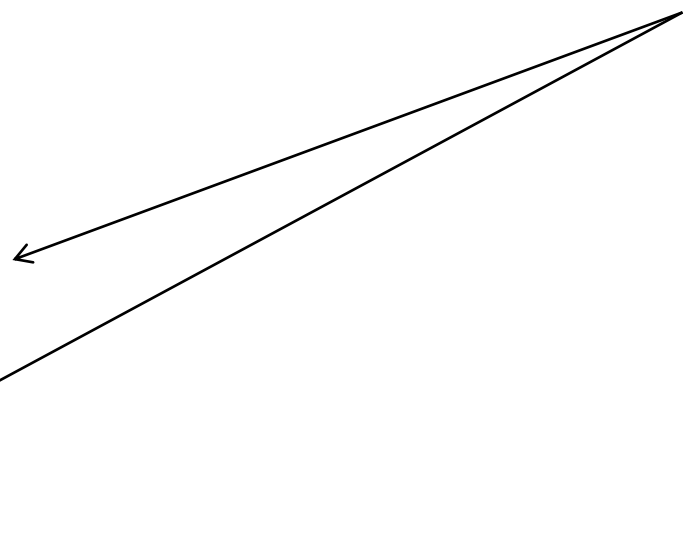
Работа с указателями

```
bool f(...)
{
    int* pt(new int);
    ...
    if (...)
    {
        delete pt;
        throw CException();
    }
    ...
    delete pt;
    return true;
}
```



Локальные переменные

```
bool f(…)
{
    CSomeClass localVar;
    ...
    if (...)
    {
        ...
        throw CException();
    }
    ...

    return true;
}
```



Деструктор класса CSomeClass

Resource acquisition is initialization

```
class CSmartWrapper
{
public:
    CSmartWrapper()
    {
        // resource acquisition
    }

    ~CSmartWrapper()
    {
         // resource releasing
    }

    // other methods
    ResourceType m_resource;
};
```

CQG

Умный указатель

```
bool f(...)
{
    CSmartPtr sPtr(new int);
    ...
    if (...)
    {
        ...
        throw CException();
    }
    ...

    return true;
}
```

```
CSmartPtr::CSmartPrt(i_ptr)
    : m_ptr(i_ptr)
{}
```

Деструктор класса CSmartPtr

```
CSmartPtr::~CSmartPrt
{
    delete m_ptr;
}
```

CSmartPtr

```
void ExampleMethod()
{
    CSmartPtr sPtr(new int);
    /*...more code...*/
}
```

Что мы хотим от CSmartPtr:

- **Чтобы с ним можно было работать как с обычным указателем.**
- **Освобождал память в деструкторе.**
- **Работал с любыми типами данных.**

```
sPtr->DoSomething();
(*sPtr).DoSomethingElse();
```

Простейшая реализация CSmartPtr

```
class CSmartPtr
{
public:
    CSmartPtr(int* i_ptr)
        : m_ptr(i_ptr)
    {}

    ~CSmartPtr()
    {
        delete m_ptr;
    }

private:
    int* m_ptr;
};
```

Простейшая реализация CSmartPtr

```
template<class T>
class CSmartPtr
{
public:
    CSmartPtr(T* i_ptr)
        : m_ptr(i_ptr)
    {}

    ~CSmartPtr()
    {
        delete m_ptr;
    }

private:
    T* m_ptr;
};
```

Простейшая реализация CSmartPtr

```
sPtr->DoSomething();
(*sPtr).DoSomethingElse();

T* operator->() const
{
   return m_ptr;
}

T& operator*() const
{
   return *m_ptr;
}
```

Улучшения

CSmartPtr sp; // default ctor

```
public:
    CSmartPtr()
        : m_ptr(nullptr)
    {
    }
```

Улучшения

```
if(sp) …
                        // приведение к bool
if(!sp) …
```

```
operator bool() const
{
    return m_ptr != nullptr;
}
```

Улучшения

```
T* p = sp.get();          // get raw pointer

T* get() const
{
    return m_ptr;
}
```

Улучшения

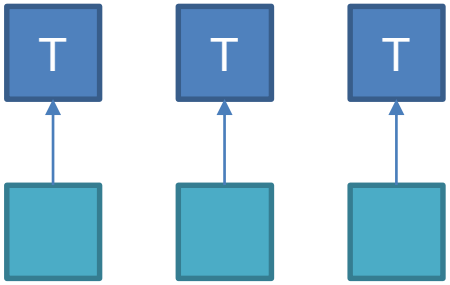
```
sp.reset();           // reset()
sp.reset(new T);      // reset(new T)


void reset()
{
    delete m_ptr;
    m_ptr = nullptr;
}

void reset(T* i_ptr)
{
    if (m_ptr != i_ptr)
    {
        delete m_ptr;
        m_ptr = i_ptr;
    }
}
```

CQG

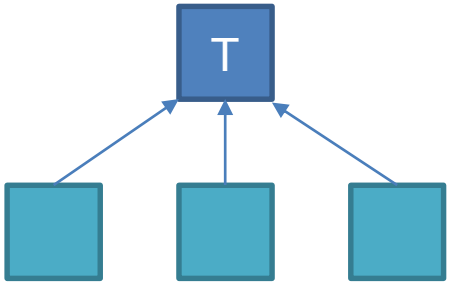
Копирование

```
SmartPtr sp1(new T);
SmartPtr sp2 = sp1;
```

Единоличное владение
std::unique_ptr



Совместное владение
std::shared_ptr



CQG

Класс CTrace

```
class CTrace
{
public:
    CTrace(int i_a) : m_a(i_a)
    {
        cout << "ctor " << m_a;
    }

    ~CTrace()
    {
        cout << "dtor " << m_a;
    }

    int m_a;
};
```

std::unique_ptr

unique_ptr. Introduction.

```
#include <memory>
using namespace std;

void g()
{
   unique_ptr<CTrace> obj(new CTrace(1)); // "ctor 1"

   cout << obj->m_a;                      // "1"

   CTrace& traceObj = *obj;
   cout << traceObj.m_a;                  // "1"
}                                         // "dtor 1"
```

Using reset

```
void g()
{
   unique_ptr<CTrace> obj(new CTrace(1)); // "ctor 1"

   obj.reset(new CTrace(2)); // "ctor 2"/"dtor 1"

   cout << obj->m_n;         // "2"

   obj.reset();              // "dtor 2"

   cout << obj->m_n;         // Access violation!
```

Using reset

```
void g()
{
   unique_ptr<CTrace> obj(new CTrace(1)); // "ctor 1"

   obj.reset(new CTrace(2)); // "ctor 2"/"dtor 1"

   cout << obj->m_n;         // "2"

   obj.reset();              // "dtor 2"

   if (obj)
   {
      cout << obj->m_n;
   }

} // Nothing
```

Передача владения

```
unique_ptr<CTrace> obj1(new CTrace(1));

unique_ptr<CTrace> obj2;
obj2 = obj1;                                  // Illegal
unique_ptr<CTrace> obj3(obj1);                // Illegal

obj2 = std::move(obj1);                       // OK
unique_ptr<CTrace> obj3(std::move(obj2)); // OK

unique_ptr<CTrace> MakePtr(...)
{
   return unique_ptr<CTrace>(new CTrace(1));
}
obj2 = MakePtr(...); // OK, implicit move operation
```

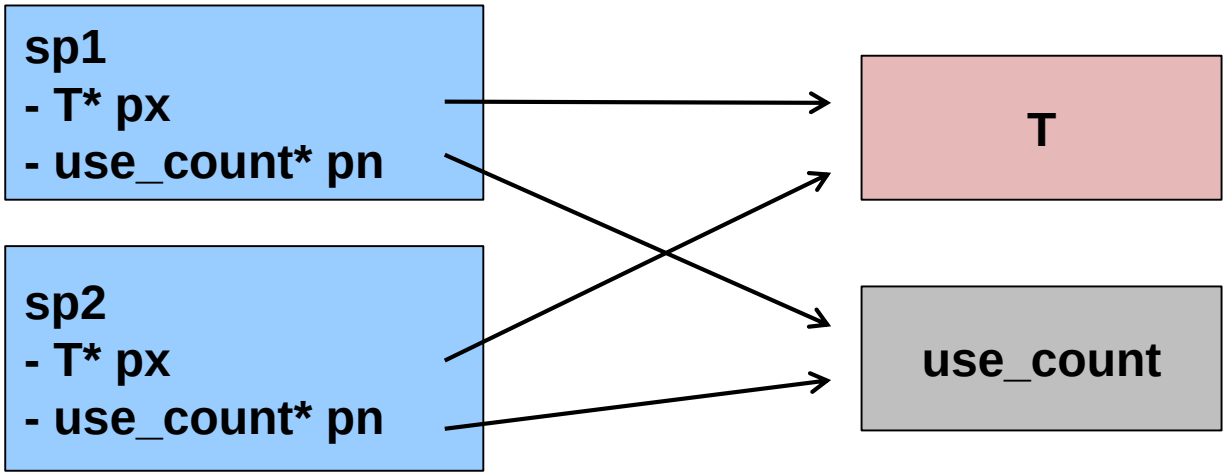
Реализация

```
template<class T>
class unique_ptr<T>
{
   …
   unique_ptr<T>(const unique_ptr<T>& i_other) = delete;
   unique_ptr<T>& operator=(const unique_ptr<T>& i_other) =
delete;

   unique_ptr<T>(unique_ptr<T>&& i_other)
      : m_ptr(i_other.m_ptr)
   {
      i_other.m_ptr = nullptr;
   }

   unique_ptr<T>& operator=(unique_ptr<T>&& i_other)
   { … }
};
```

std::shared_ptr

shared_ptr

```
shared_ptr<T> sp1(new T);      // use_count = 1
shared_ptr<T> sp2 = sp1;       // use_count = 2
```

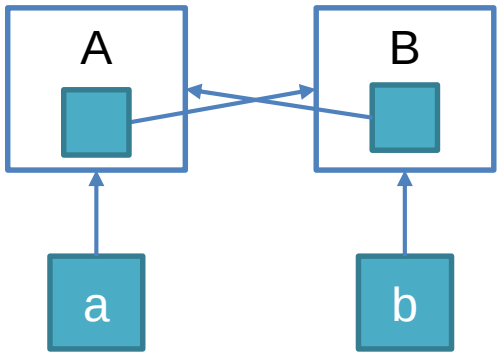


shared_ptr

```
shared_ptr<CTrace> f()                                  use_count
{
   shared_ptr<CTrace> sPt1(new CTrace(1));                  1
   // "ctor 1"

   shared_ptr<CTrace> sPt2 = sPt1;                          2
   // sPt1.get() == sPt2.get()

   sPt1.reset();                                            1
   return sPt2;
}

void g()
{
   shared_ptr<CTrace> sPt3(f());                            1
} // "dtor 1"                                               0
```

shared_ptr

```
class A
{
    shared_ptr<B> m_ptr;
};
class B
{
    shared_ptr<A> m_ptr;
};
void func()
{
    shared_ptr<A> a(new A);
    shared_ptr<B> b(new B);
    a->m_ptr = b;
    b->m_ptr = a;
    // use_count = 2
} // use_count = 1
```



weak_ptr

weak_ptr – указатель, который так же как и shared_ptr связан со счетчиком ссылок, однако он не увеличивает счетчика ссылок. weak_ptr может быть создан только из shared_ptr.

```
shared_ptr<CTrace> sPtr1(new CTrace(1));
weak_ptr<CTrace> sPtr2(sPtr1);
...
cout << sPtr2->m_n; // Illegal
```

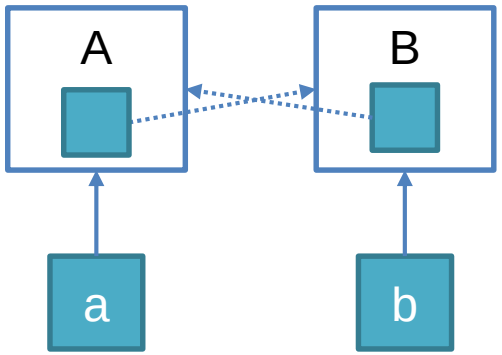
weak_ptr

weak_ptr – указатель, который так же как и shared_ptr связан со счетчиком ссылок, однако он не увеличивает счетчика ссылок. weak_ptr может быть создан только из shared_ptr.

```
shared_ptr<CTrace> sPtr1(new CTrace(1));
weak_ptr<CTrace> sPtr2(sPtr1);
...
shared_ptr<CTrace> sPtr3 = sPtr2.lock();
if (sPtr3)
{
    cout << sPtr3->m_n;   // “1”
}
else
{
    cout << “Error”;
}
```

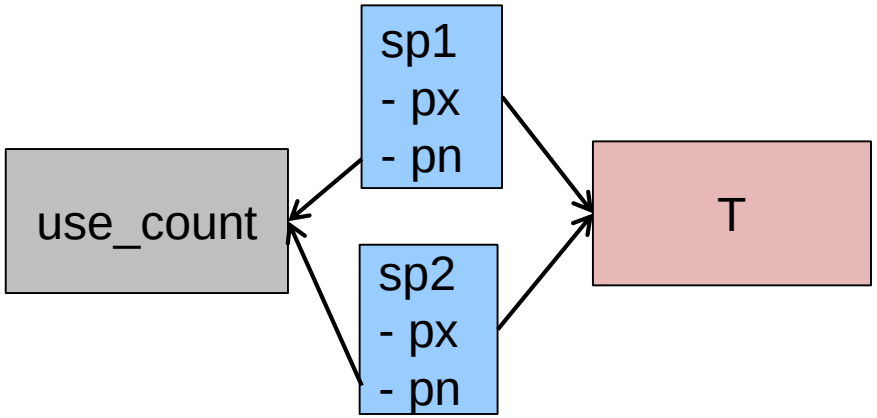
shared_ptr

```
class A
{
    weak_ptr<B> m_ptr;
};
class B
{
    weak_ptr<A> m_ptr;
}
void func()
{
    shared_ptr<A> a(new A);
    shared_ptr<B> b(new B);
    a->m_ptr = b;
    b->m_ptr = a;
}
```



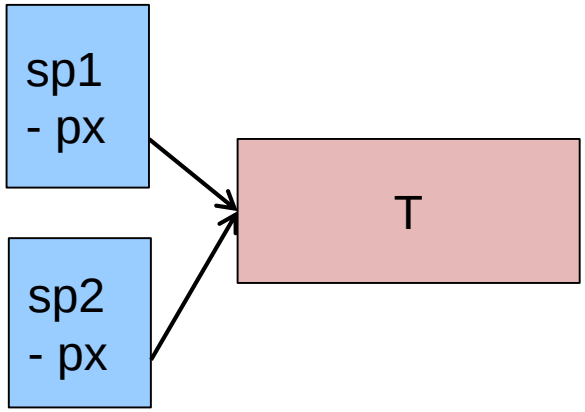
shared_ptr vs unique_ptr vs raw ptr

shared_ptr



+ Можно копировать
– Размер = 2 raw указателя
– Копировать дорого

unique_ptr



+ Размер = 1 raw указатель
– Нельзя копировать

Другие умные указатели

- `std::auto_ptr` –
  неявная передача владения при копировании (deprecated).

- `boost::scoped_ptr` –
  запрещены copy/move.

- `boost::intrusive_ptr` –
  для объектов со встроенным счетчиком ссылок.

Smart Pointers. Best practices

- **Не используйте raw pointer для хранения объектов**

  MyClass* a = new MyClass(…); // BAD PRACTICE!
  shared_ptr<MyClass> sPtr = new MyClass(…);

- **Не создавайте умные указатели из обычных указателей**
  Here be dragons!

- **Используйте make_shared() для создания shared_ptr**
- **Используйте make_unique() для создания unique_ptr**

  auto sPtr = make_shared<MyClass>(…);
  auto sPtr = make_unique<MyClass>(…);

  это короче, безопаснее, оптимальнее (для shared).

CQG

Smart Pointers. Best practices

- **Не используйте умные указатели без необходимости**

```
void print(A* a) // good
{
   cout << a->m_data << endl;
};

void print(A& a) { ... } // better

void print(shared_ptr<A> a) { ... } // BAD
void print(unique_ptr<A> a) { ... } // BAD
```

Bonus: nullptr

nullptr – константа нулевого указателя

Не используйте 0 или NULL для инициализации пустых указателей.

Bonus: noexcept

noexcept – обещание не бросать исключения из функции
Если исключение будет брошено, программа падает

```
void foo();
// Может бросить исключение, нужно с этим что-то делать

void bar() noexcept;
// Не может бросить исключение
// А если и бросит, то делать с этим ничего не нужно
// Все равно программа упадет
```

Bonus: delete functions

```
struct A
{
    int m_data;
}


A a; // OK
A b(a); // OK, copy it
```

Bonus: delete functions

```
struct A
{
    int m_data;
    A(const A&) = delete;
}

A a; // OK
A b(a); // Error!
```

Bonus: default functions

```
class A
{
   int m_data;
public:
   A(int data) : m_data(data) {}
}


A a(10); // OK
A b; // Error!
```

Bonus: default functions

```
class A
{
    int m_data;
public:
    A(int data) : m_data(data) {}
    A() = default;
}

A a(10); // OK
A b; // OK
```

Литература по теме:

1. **Meyers S., Effective Modern C++ (Best choice!)**
2. Karlsson B., Beyond the C++ Standard
3. Sutter H., Exceptional C++
3. Sutter H., More Exceptional C++
4. http://habrahabr.ru/post/226229/
5. http://habrahabr.ru/post/242639/
6. http://stackoverflow.com/questions/3106110/what-are-move-semantics
7. http://msdn.microsoft.com/ru-ru/library/hh279676.aspx
8. http://habrahabr.ru/post/140222/


Buy
100
8119
LMT DAY
Sell 100
8119
STP DAY
Sell 100
8119
LMT DAY
Buy 100@8119 LMT DAY

Q&A