



Modern C++

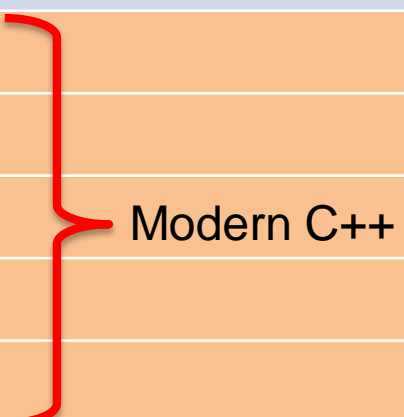
Move Semantic. Smart Pointers.

Ращенко Елена

06.10.2021

История версий C++

1972	C
1980	C with classes
1983	C++
1985	“The C++ programming language”
1998	C++98, STL
1999	Boost
2003	C++03
2011	C++11
2014	C++14
2017	C++17
2020	C++20
	...



Modern C++



Move Semantic

Move Semantic

1. Работа с большими объектами в старом C++
2. Return Value Optimization/Named Return Value Optimization
3. Как устроен вектор?
4. lvalue и rvalue
5. lvalue reference
6. Move конструктор и move оператора присваивания
7. `std::move`
8. Ключевое слово `default`
9. Ключевое слово `delete`
10. Спецификатор `noexcept`

Передача больших объектов в функцию

```
void ProcessData(std::vector<int> i_data)
{
    ...
    ...
}
```

```
int main(...)
{
    std::vector<int> veryBigVector = ...;
    ...
    ProcessData(veryBigVector); // vector будет скопирован
    ...
}
```

Передача больших объектов в функцию

```
void ProcessData(const std::vector<int>& i_data)
{
    ...
    ...
}
```

```
int main(...)
{
    std::vector<int> veryBigVector = ...;
    ...
    ProcessData(veryBigVector); // vector НЕ будет скопирован
    ...
}
```

Передача больших объектов в функцию

```
void ProcessData(const std::vector<int>& i_data)
{
    vector<int> data = i_data; // vector будет скопирован
    ... // обработка данных
}
```

```
int main(...)
{
    std::vector<int> veryBigVector = ...;
    ...
    ProcessData(veryBigVector); // vector НЕ будет скопирован
    ...
}
```


Передача больших объектов в функцию

```
void X::SetData (const std::vector<int>& i_data)
{
    m_data = i_data; // vector будет скопирован
}
```

```
int main(...)
{
    std::vector<int> veryBigVector = ...;
    X a;
    a.SetData(veryBigVector); // vector НЕ будет скопирован
    ...
}
```


Проверка данных перед сохранением

```
void X::UpdateData(...)
{
    std::vector<int> temporaryVector = ...;
    if (verifyData(temporaryVector))
    {
        m_data = temporaryVector; // vector будет скопирован
    }
    ...
}
```

Возврат больших объектов из функции

```
std::vector<int> GetData()  
{  
    std::vector<int> bigVector = ...;  
    ...  
    return bigVector; // копирование из bigVector в возвращаемое значение  
}
```

```
int main(...)  
{  
    std::vector<int> bigVectorCopy = GetData(); // копирование из  
                                                // возвращаемого значения  
                                                // в bigVectorCopy  
    ...  
}
```

Возврат больших объектов из функции

```
std::vector<int> GetData()  
{  
    std::vector<int> bigVector = ...;  
    ...  
    return bigVector; // RVO или конструктор копирования  
}
```

```
int main(...)  
{  
    std::vector<int> bigVectorCopy = GetData(); // копирования точно не  
                                                // будет начиная с C++17  
                                                // RVO  
    ...  
}
```

Возврат больших объектов из функции

```
void GetData(std::vector<int>& o_data)
{
    ...
    ...
    o_data = ...;
}
```

```
int main(...)
{
    std::vector<int> bigVector;
    GetData(veryBigVector); // копирования не будет
    ...
    ...
}
```

Return value optimization

```
struct X
{
    X() { cout << "X()" << endl; }
    X(const X&) { cout << "X(X const&)" << endl; }
}
```

Сколько копирований произойдет при вызове этих функций?

```
X foo()
{
    ...
    return X();
}
```

Return value optimization

```
struct X
{
    X() { cout << "X()" << endl; }
    X(const X&) { cout << "X(X const&)" << endl; }
}
```

Сколько копирований произойдет при вызове этих функций?

```
X foo()
{
    ...
    return X();
}
```

- 1
- 0 при RVO

Return value optimization

```
struct X
{
    X() { cout << "X()" << endl; }
    X(const X&) { cout << "X(X const&)" << endl; }
}
```

Сколько копирований произойдет при вызове этих функций?

```
X foo()
{
    ...
    return X();
}
```

```
X bar()
{
    X value;
    ...
    return value;
}
```

- 1
- 0 при RVO

Return value optimization

```
struct X
{
    X() { cout << "X()" << endl; }
    X(const X&) { cout << "X(X const&)" << endl; }
}
```

Сколько копирований произойдет при вызове этих функций?

```
X foo()
{
    ...
    return X();
}
```

```
X bar()
{
    X value;
    ...
    return value;
}
```

- 1
- 0 при RVO

- 1
- 0 при RVO

Return value optimization

```
struct X
{
    X() { cout << "X()" << endl; }
    X(const X&) { cout << "X(X const&)" << endl; }
}
```

Сколько копирований произойдет при вызове этих функций?

```
X foo()
{
    ...
    return X();
}
```

```
X bar()
{
    X value;
    ...
    return value;
}
```

```
X rol()
{
    X value1;
    X value2;
    bool cond = true;
    ...
    return cond ? value1 : value2;
}
```

- 1
- 0 при RVO

- 1
- 0 при RVO

Return value optimization

```
struct X
{
    X() { cout << "X()" << endl; }
    X(const X&) { cout << "X(X const&)" << endl; }
}
```

Сколько копирований произойдет при вызове этих функций?

```
X foo()
{
    ...
    return X();
}
```

```
X bar()
{
    X value;
    ...
    return value;
}
```

```
X rol()
{
    X value1;
    X value2;
    bool cond = true;
    ...
    return cond ? value1 : value2;
}
```

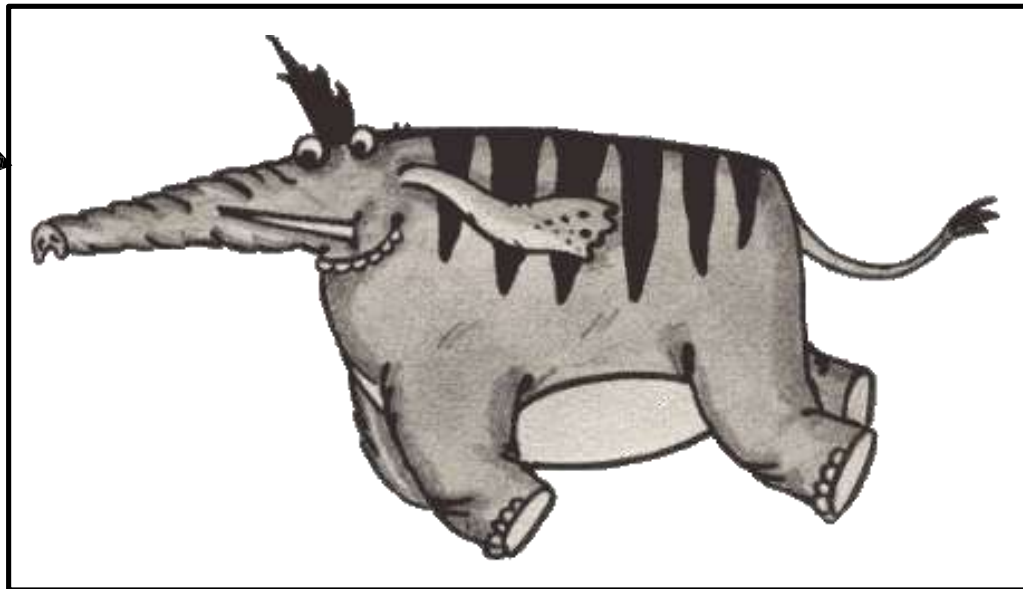
- 1
- 0 при RVO

- 1
- 0 при RVO

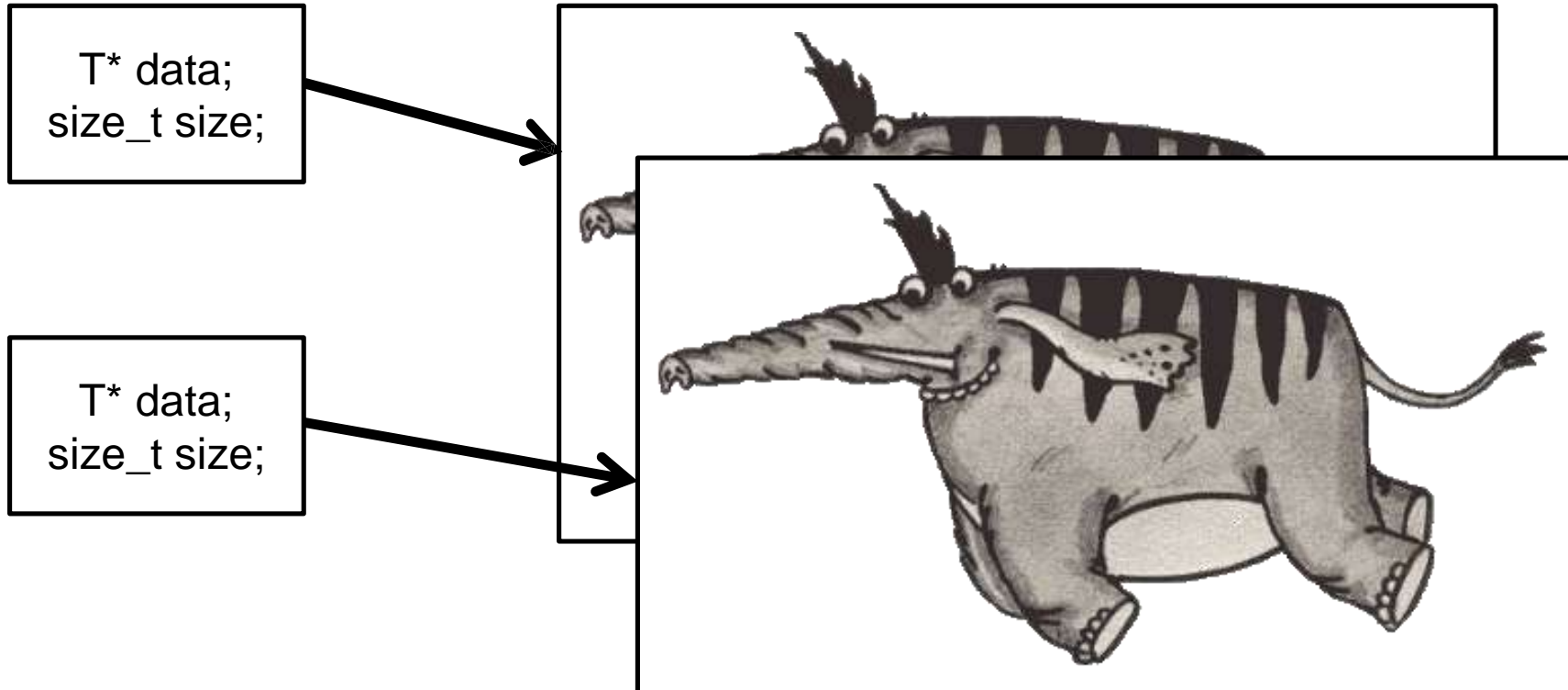
- 1

Что такое вектор?

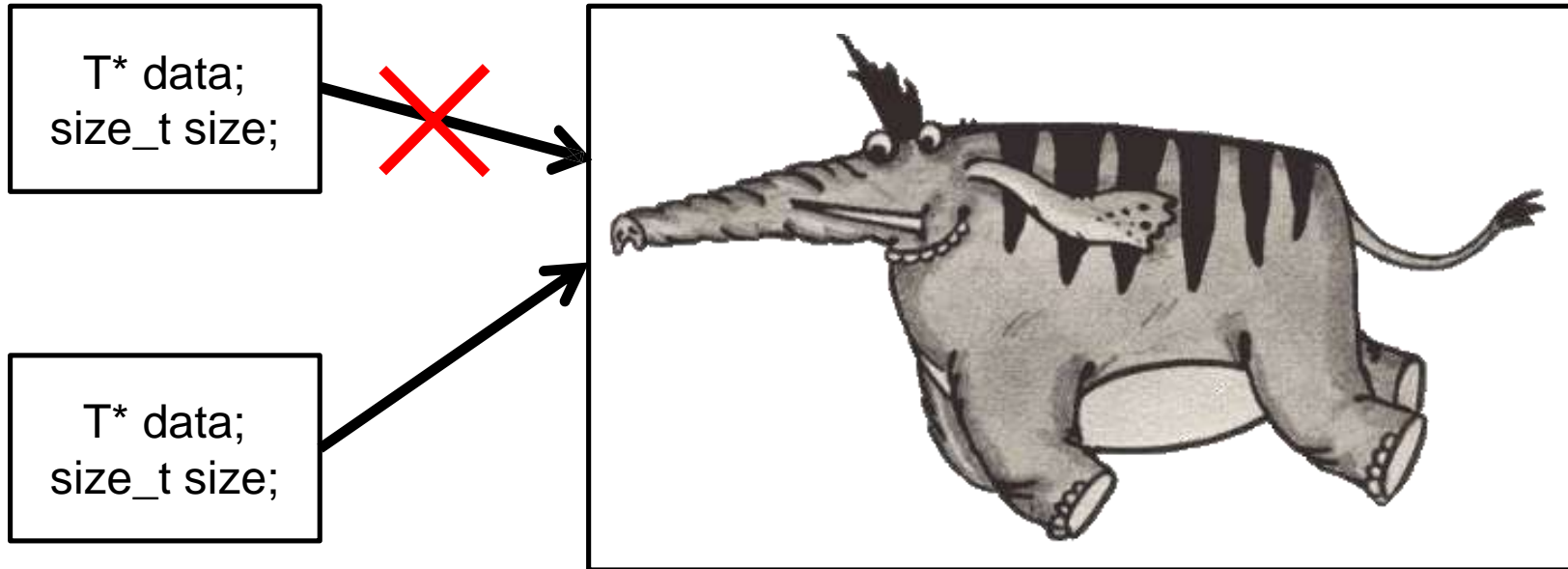
T* data;
size_t size;



Копирование вектора



Перемещение вектора





Move semantic

Необходимые и достаточные условия при которых произойдет перемещение:

1. Компилятор знает, что этот объект **можно перемещать** (компилятор должен уметь отличать временный объект от невременного)
2. Компилятор знает **как перемещать** объект данного типа (определен move конструктор, move оператор присваивания)

Lvalue vs Rvalue

Lvalue

Выражение, описывающее **не временный** объект, чаще всего именованный

У этого объекта можно взять адрес

То, что может быть слева от знака '=' (если объект неконстантный)

Rvalue

Выражение, описывающее **временный объект** или return выражение

То, что может быть справа от знака '='

Lvalue vs Rvalue

Lvalue

```
string str = "foo";  
str = "bar";
```

Rvalue

Lvalue vs Rvalue

Lvalue

```
string str = "foo";  
str = "bar";
```

Rvalue

```
string str = "foo";  
"foo" = ... // error
```

Lvalue vs Rvalue

Lvalue

```
string str = "foo";  
str = "bar";
```

```
int a = 5;  
int b = a + 3;  
int c = a + b;
```

Rvalue

```
string str = "foo";  
"foo" = ... // error
```

Lvalue vs Rvalue

Lvalue

```
string str = "foo";  
str = "bar";
```

```
int a = 5;  
int b = a + 3;  
int c = a + b;
```

Rvalue

```
string str = "foo";  
"foo" = ... // error
```

```
int c = a + b;  
a + 3 = ... // error  
a + b = ... // error
```

Lvalue vs Rvalue

Lvalue

```
string str = "foo";  
str = "bar";
```

```
int a = 5;  
int b = a + 3;  
int c = a + b;
```

```
int& getData = { return a; }  
getData() = 10;
```

Rvalue

```
string str = "foo";  
"foo" = ... // error
```

```
int c = a + b;  
a + 3 = ... // error  
a + b = ... // error
```

Lvalue vs Rvalue

Lvalue

```
string str = "foo";  
str = "bar";
```

```
int a = 5;  
int b = a + 3;  
int c = a + b;
```

```
int& getData = { return a; }  
getData() = 10;
```

Rvalue

```
string str = "foo";  
"foo" = ... // error
```

```
int c = a + b;  
a + 3 = ... // error  
a + b = ... // error
```

```
int getData = { return a; }  
getData() = 10; // error
```


Lvalue vs Rvalue

Lvalue

```
string str = "foo";  
str = "bar";
```

```
int a = 5;  
int b = a + 3;  
int c = a + b;
```

```
int& getData = { return a; }  
getData() = 10;
```

```
const int d = 10;  
d = 7; // error
```

Rvalue

```
string str = "foo";  
"foo" = ... // error
```

```
int c = a + b;  
a + 3 = ... // error  
a + b = ... // error
```

```
int getData = { return a; }  
getData() = 10; // error
```

Rvalue ссылка &&

- 1) T& ссылка на изменяемый объект, lvalue ссылка
- 2) const T& ссылка на неизменяемый объект, const lvalue ссылка
- 3) T&& ссылка на **временный** объект, **rvalue** ссылка

- 1) void foo(string& s)
- 2) void foo(const string& s)
- 3) void foo(string&& s)

В большинстве случаев rvalue ссылки используются только в параметрах функции.

Как “научить” класс перемещаться

```
class A
{
public:
    A(); // конструктор по умолчанию
    A(int); // конструктор с параметром

};
```

Как “научить” класс перемещаться

```
class A
{
public:
    A(); // конструктор по умолчанию
    A(int); // конструктор с параметром

    A(const A&); // copy конструктор
    A& operator=(const A&); // copy оператор присваивания

};
```

Как “научить” класс перемещаться

```
class A
{
public:
    A(); // конструктор по умолчанию
    A(int); // конструктор с параметром

    A(const A&); // copy конструктор
    A& operator=(const A&); // copy оператор присваивания

    A(A&&); // move конструктор
    A& operator=(A&&); // move оператор присваивания
};
```

Пример: IntArray

```
class IntArray
{
private:
    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; }
};
```

Пример: IntArray

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


Пример: IntArray

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

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

Пример: IntArray

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

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

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

Пример: IntArray

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

```
// copy оператор присваивания
IntArray& operator=(const IntArray& i_other)
{
    if (this == &i_other) return *this;
    delete[] m_data;
    m_size = i_other.m_size;
    m_data = new int[m_size];
    // долго
    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) // быстро
{
    i_other.m_data = nullptr;
    i_other.m_size = 0;
}
```

```
// move оператор присваивания
IntArray& operator=(IntArray&& i_other)
{
    if (this == &i_other) return *this;
    delete[] m_data;
    m_size = i_other.m_size;
    m_data = i_other.m_data; // быстро
    i_other.m_data = nullptr;
    i_other.m_size = 0;
    return *this;
}
```

Пример работы IntArray

```
IntArray GetData(int i_size)
{
    IntArray data(i_size);
    return data;
}

int main()
{
    IntArray a1(1000);

    IntArray a2(a1);           // конструктор копирования (a1 - lvalue)      (1)

}
```

Пример работы IntArray

```
IntArray GetData(int i_size)
{
    IntArray data(i_size);
    return data;           // либо RVO, либо move конструктор
}

int main()
{
    IntArray a1(1000);

    IntArray a2(a1);        // конструктор копирования (a1 - lvalue)      (1)
    IntArray a3 = GetData(2000); // RVO                                  (2)

}
```

Пример работы IntArray

```
IntArray GetData(int i_size)
{
    IntArray data(i_size);
    return data;           // либо NRVO, либо move конструктор
}

int main()
{
    IntArray a1(1000);

    IntArray a2(a1);        // конструктор копирования (a1 - lvalue)           (1)
    IntArray a3 = GetData(2000); // RVO                                           (2)

    a2 = a3;                // copy оператор присваивания (a3 - lvalue)         (3)
}
```

Пример работы IntArray

```
IntArray GetData(int i_size)
{
    IntArray data(i_size);
    return data;           // либо NRVO, либо move конструктор
}

int main()
{
    IntArray a1(1000);

    IntArray a2(a1);        // конструктор копирования (a1 - lvalue)           (1)
    IntArray a3 = GetData(2000); // RVO                                           (2)

    a2 = a3;               // copy оператор присваивания (a3 - lvalue)           (3)
    a2 = GetData(500);     // move оператор присваивания (GetData(500) - rvalue) (4)

}
```


Пример работы IntArray

```
IntArray GetData(int i_size)
{
    IntArray data(i_size);
    return data;           // либо NRVO, либо move конструктор
}

int main()
{
    IntArray a1(1000);

    IntArray a2(a1);        // конструктор копирования (a1 - lvalue)           (1)
    IntArray a3 = GetData(2000); // RV0                                       (2)

    a2 = a3;               // copy оператор присваивания (a3 - lvalue)         (3)
    a2 = GetData(500);     // move оператор присваивания (GetData(500) - rvalue) (4)

    IntArray a4(100);
    IntArray a5(200);
    ...
    IntArray a6(a4);        // конструктор копирования (a4 - lvalue)           (5)
    a2 = a5;               // copy оператор присваивания (a5 - lvalue)         (6)
}
```



std::move

- Как объяснить компилятору, что объект все-таки временный?

std::move

- Как объяснить компилятору, что объект все-таки временный?
- Использовать std::move



std::move

- Как объяснить компилятору, что объект все-таки временный?
- Использовать std::move
- std::move не перемещает объект, а только изменяет его тип с lvalue на rvalue



std::move

- Как объяснить компилятору, что объект все-таки временный?
- Использовать std::move
- std::move не перемещает объект, а только изменяет его тип с lvalue на rvalue
- После передачи объекта в функцию по &&-ссылке его использовать нельзя

std::move

- Как объяснить компилятору, что объект все-таки временный?
- Использовать std::move
- std::move не перемещает объект, а только изменяет его тип с lvalue на rvalue
- После передачи объекта в функцию по &&-ссылке его использовать нельзя

```
IntArray a4(100);  
IntArray a5(200);  
  
IntArray a6(a4);           // конструктор копирования (a4 - lvalue)      (1)  
a2 = a5;                  // copy оператор присваивания (a5 - lvalue)    (2)  
  
IntArray a7(std::move(a4)); // move конструктор                          (3)  
a2 = std::move(a5);        // move оператор присваивания                    (4)
```

Rvalue ссылка – rvalue?

- Всегда ли rvalue ссылка – это rvalue? Нет, не всегда.
- Именованная rvalue ссылка - это lvalue.
- Как ее передать дальше как rvalue?

```
void SendData(IntArray&& i_data)
{
    ...
}

void ProcessData(IntArray&& i_data)
{
    SendData(i_data);           // illegal
}

int main()
{
    IntArray a1(1000);
    ProcessData(std::move(a1));
}
```


Rvalue ссылка – rvalue?

- Всегда ли rvalue ссылка – это rvalue? Нет, не всегда.
- Именованная rvalue ссылка - это lvalue.
- Как ее передать дальше как rvalue?
- Использовать `std::move`.

```
void SendData(IntArray&& i_data)
{
    ...
}

void ProcessData(IntArray&& i_data)
{
    SendData(std::move(i_data));
}

int main()
{
    IntArray a1(1000);
    ProcessData(std::move(a1));
}
```

Автогенерация move конструктора

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

- Все поля класса класса могут быть перемещены
- Все базовые классы умеют перемещаться
- Нет деструктора
- Нет сору конструктора/сору оператора присваивания
- Нет move конструктора/move оператора присваивания

Автогенерация move конструктора

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

- Все поля класса класса могут быть перемещены
- Все базовые классы умеют перемещаться
- Нет деструктора
- Нет сору конструктора/сору оператора присваивания
- Нет move конструктора/move оператора присваивания

2) Move конструктор и move оператор присваивания “отключают” автоматическую генерацию других конструкторов и операторов присваивания.



Ключевое слово default

- Говорит компилятору самостоятельно генерировать функцию класса (если она не объявлена в классе)

Ключевое слово default

- Говорит компилятору самостоятельно генерировать функцию класса (если она не объявлена в классе)
- Может применяться к
 - конструктору по умолчанию
 - конструктору копирования
 - move конструктору
 - сору оператору присваивания
 - move оператору присваивания
 - деструктору

Ключевое слово default

- Говорит компилятору самостоятельно генерировать функцию класса (если она не объявлена в классе)
- Может применяться к
 - конструктору по умолчанию
 - конструктору копирования
 - move конструктору
 - сору оператору присваивания
 - move оператору присваивания
 - деструктору

```
class X
{
public:
    ~X() { ... }

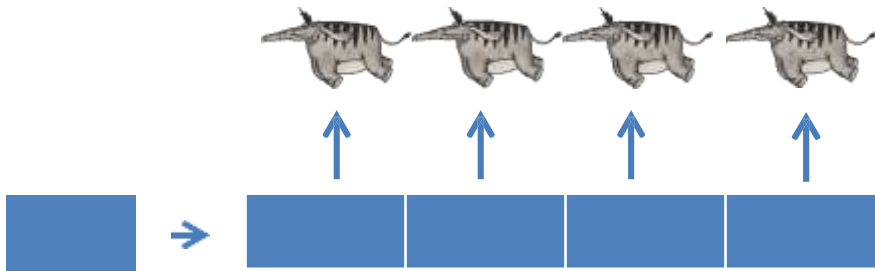
    X(X&& x) = default;
};
```

vector of vectors

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

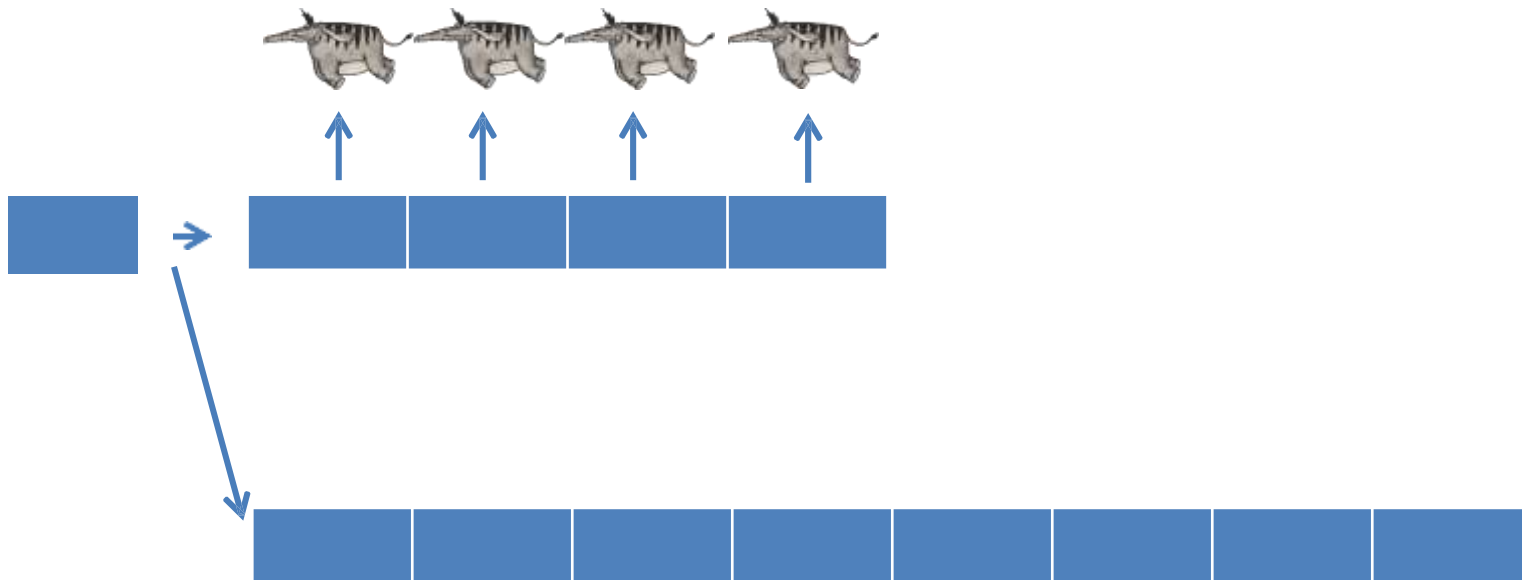

vector of vectors

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



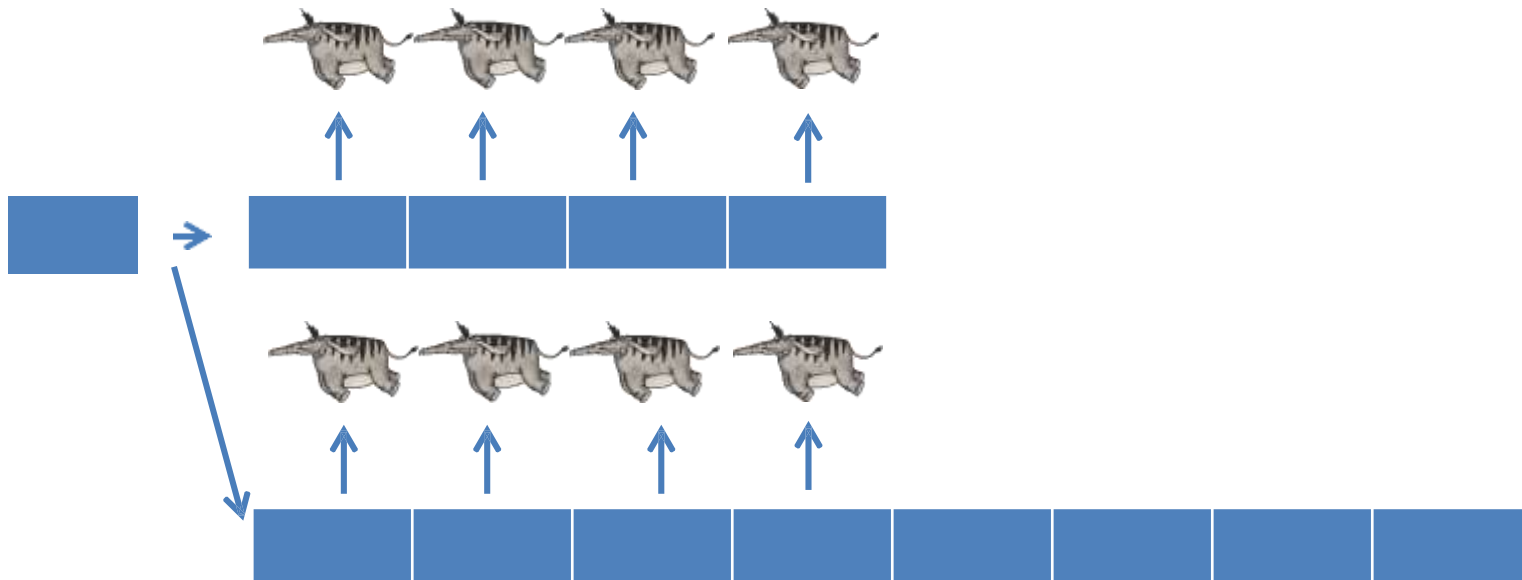
vector of vectors

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



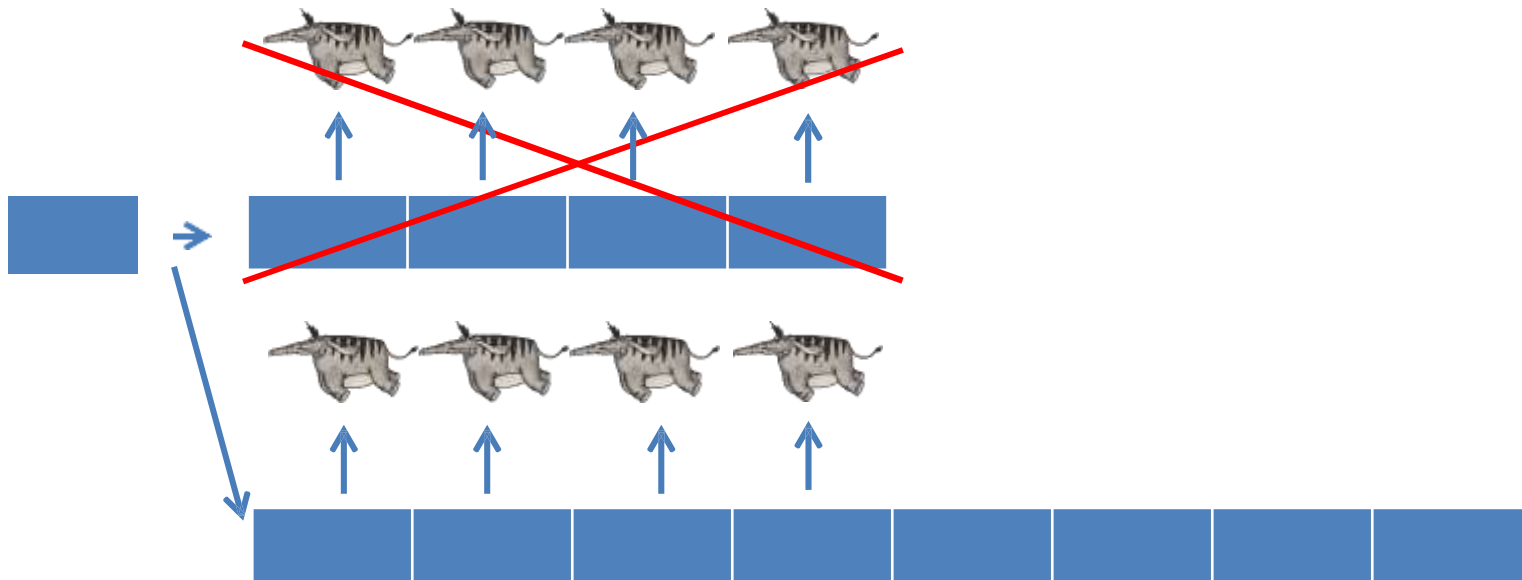
vector of vectors

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



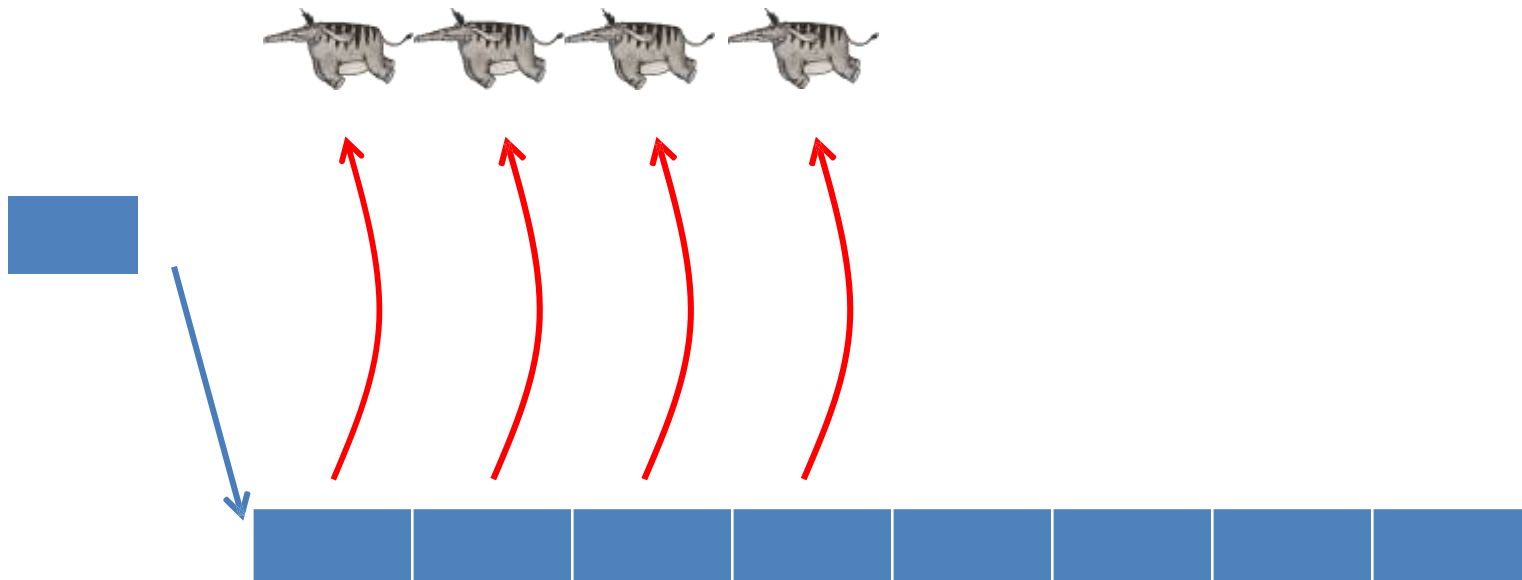
vector of vectors

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



vector of vectors

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



Спецификатор поехсерт

- Для того, чтобы стандартные контейнеры вроде `vector<T>` могли использовать семантику перемещения, хранящиеся в контейнере объекты должны иметь `move`-конструктор и `move`-оператор присваивания, которые не кидают исключения.

Спецификатор noexcept

- Для того, чтобы стандартные контейнеры вроде `vector<T>` могли использовать семантику перемещения, хранящиеся в контейнере объекты должны иметь `move`-конструктор и `move`-оператор присваивания, которые не кидают исключения.
- Спецификатор **noexcept** говорит компилятору, что функция не будет выбрасывать исключения.

```
class X
{
    X(X&&) noexcept;
    X& operator=(X&&) noexcept;
};
```


Спецификатор noexcept

- Для того, чтобы стандартные контейнеры вроде `vector<T>` могли использовать семантику перемещения, хранящиеся в контейнере объекты должны иметь `move-конструктор` и `move-оператор присваивания`, которые не кидают исключения.
- Спецификатор `noexcept` говорит компилятору, что функция не будет выбрасывать исключения.

```
class X
{
    X(X&&) noexcept;
    X& operator=(X&&) noexcept;
};
```

- На самом деле `noexcept` не запрещает функции выбрасывать исключения.

Спецификатор noexcept

- Для того, чтобы стандартные контейнеры вроде `vector<T>` могли использовать семантику перемещения, хранящиеся в контейнере объекты должны иметь `move-конструктор` и `move-оператор присваивания`, которые не кидают исключения.
- Спецификатор `noexcept` говорит компилятору, что функция не будет выбрасывать исключения.

```
class X
{
    X(X&&) noexcept;
    X& operator=(X&&) noexcept;
};
```

- На самом деле `noexcept` не запрещает функции выбрасывать исключения.
- при возникновении исключения, если оно происходит из `noexcept-функции`, будет вызвана функция `std::terminate()`.



Вопросы?





Smart Pointers

1. Работа с указателями в старом C++
2. Идиома RAII
3. CSmartPtr
4. `std::unique_ptr`
5. `std::shared_ptr`
6. `std::weak_ptr`
7. Советы по использованию умных указателей
8. `std::make_unique()`, `std::make_shared`

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;  
}
```

?



Работа с указателями

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

Разрушение
локальных
переменных

Локальные переменные

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

Деструктор
класса
CSomeClass

Умный указатель

```
bool f(...)
{
    CSharedPtr sp(new int);

    ...
    if (...)
    {
        ...
        throw CException();
    }
    ...
    return true;
}
```


Умный указатель

```
bool f(...)
{
    CSmartPtr sp(new int);

    ...
    if (...)
    {
        ...
        throw CException();
    }
    ...
    ...
    return true;
}
```

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

Умный указатель

```
bool f(...)
{
    CSmartPtr sp(new int);

    ...
    if (...)
    {
        ...
        throw CException();
    }
    ...
    ...
    return true;
}
```

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

**Деструктор
класса
CSmartPtr**

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

Идиома RAI

```
class CSmartWrapper
{
public:
    CSmartWrapper()
    {
        // получение ресурса
    }

    ~CSmartWrapper()
    {
        // освобождение ресурса
    }
    ResourceType m_resource;
};
```

Resource Acquisition Is Initialization

Получение ресурса есть
инициализация

CSmartPointer

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

    ~CSmartPointer()
    {
        delete m_ptr;
    }

private:
    int* m_ptr;
};
```

Что мы хотим от CSmartPointer:

- Освобождал память в деструкторе

CSharedPtr

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

    ~CSharedPtr()
    {
        delete m_ptr;
    }

private:
    T* m_ptr;
};
```

Что мы хотим от CSharedPtr:

- Освобождал память в деструкторе
- Работал с любыми типами данных

CSharedPtr

```
template<class T>
class CSharedPtr
{
public:
    T* operator->() const
    {
        return m_ptr;
    }

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

Что мы хотим от CSharedPtr:

- Освобождал память в деструкторе
- Работал с любыми типами данных
- Можно работать как с raw указателем
`sPtr->DoSomething();`
`(*sPtr).DoSomethingElse();`

CSmartPointer

```
template<class T>
class CSmartPointer
{
public:
    CSmartPointer()
        : m_ptr(nullptr)
    {}

    // nullptr используется для
    // инициализации нулевых
    // указателей

private:
    T* m_ptr;
};
```

Что мы хотим от CSmartPointer:

- Освобождал память в деструкторе
- Работал с любыми типами данных
- Можно работать как с raw указателем
sPtr->DoSomething();
(*sPtr).DoSomethingElse();
- Конструктор по умолчанию
CSmartPointer sp;

CSharedPtr

```
template<class T>
class CSharedPtr
{
public:
    CSharedPtr()
        : m_ptr(nullptr)
    {}

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

private:
    T* m_ptr;
};
```

Что мы хотим от CSharedPtr:

- Освобождал память в деструкторе
- Работал с любыми типами данных
- Можно работать как с raw указателем
sPtr->DoSomething();
(*sPtr).DoSomethingElse();
- Конструктор по умолчанию
CSharedPtr sp;
- Приведение к bool
if(sp) / if(!sp)

CSmartPointer

```
template<class T>
class CSmartPointer
{
public:
    T* get() const
    {
        return m_ptr;
    }

private:
    T* m_ptr;
};
```

Что мы хотим от CSmartPointer:

- Освобождал память в деструкторе
- Работал с любыми типами данных
- Можно работать как с raw указателем
sPtr->DoSomething();
(*sPtr).DoSomethingElse();
- Конструктор по умолчанию
CSmartPointer sp;
- Приведение к bool
if(sp) / if(!sp)
- Получение raw указателя
T* p = sp.get();

CSharedPtr

```
template<class T>
class CSharedPtr
{
public:
    void reset(
        T* i_ptr = nullptr)
    {
        if (m_ptr != i_ptr)
        {
            delete m_ptr;
            m_ptr = i_ptr;
        }
    }
private:
    T* m_ptr;
};
```

Что мы хотим от CSharedPtr:

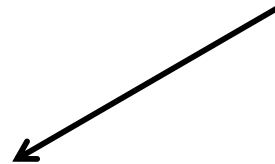
- Освобождал память в деструкторе
- Работал с любыми типами данных
- Можно работать как с raw указателем
sPtr->DoSomething();
(*sPtr).DoSomethingElse();
- Конструктор по умолчанию
CSharedPtr sp;
- Приведение к bool
if(sp) / if(!sp)
- Получение raw указателя
T* p = sp.get();
- Удаление содержимого **sp.reset();**
Замена содержимого **sp.reset(new T);**

Копирование умных указателей

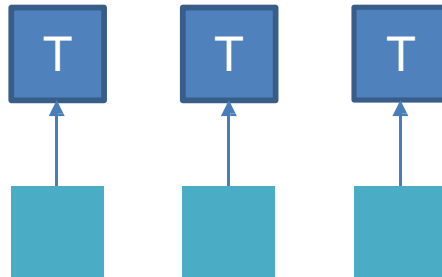
```
SmartPtr sp1(new T);  
SmartPtr sp2 = sp1;
```

Копирование умных указателей

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



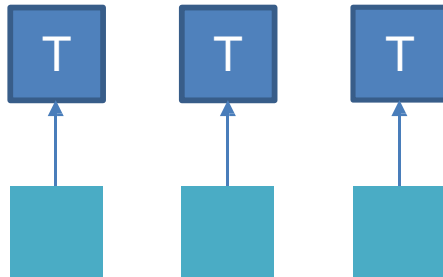
Единоличное
владение
std::unique_ptr



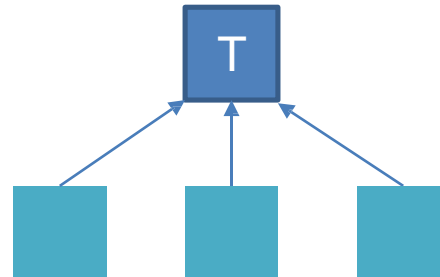
Копирование умных указателей

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

**Единоличное
владение**
std::unique_ptr



**Совместное
владение**
std::shared_ptr



Класс CTrace

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

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

    int m_a;
};
```




`std::unique_ptr`

- Владеет объектом эксклюзивно
- Нельзя копировать, но можно перемещать (`std::move`)

Пример работы `std::unique_ptr`

```
void foo
{
    std::unique_ptr<CTrace> sp(new CTrace(1)); // “ctor 1”
```

Пример работы std::unique_ptr

```
void foo
{
    std::unique_ptr<CTrace> sp(new CTrace(1)); // “ctor 1”

    std::cout << sp->m_a;                      // operator->, “1”
}
```

Пример работы std::unique_ptr

```
void foo
{
    std::unique_ptr<CTrace> sp(new CTrace(1)); // “ctor 1”

    std::cout << sp->m_a;                      // operator->, “1”

    CTrace& traceObj = *sp;                    // operator*
    std::cout << traceObj.m_a;                 // “1”
}
```

Пример работы std::unique_ptr

```
void foo
{
    std::unique_ptr<CTrace> sp(new CTrace(1)); // “ctor 1”

    std::cout << sp->m_a;                      // operator->, “1”

    CTrace& traceObj = *sp;                    // operator*
    std::cout << traceObj.m_a;                 // “1”
}                                              // “dtor 1”
```

Пример работы std::unique_ptr

```
void foo
{
    std::unique_ptr<CTrace> sp(new CTrace(1)); // “ctor 1”

    sp.reset(new CTrace(2));                 // “ctor 2”/“dtor 1”
    std::cout << sp->m_a;                     // “2”
}
```

Пример работы std::unique_ptr

```
void foo
{
    std::unique_ptr<CTrace> sp(new CTrace(1)); // “ctor 1”

    sp.reset(new CTrace(2));                  // “ctor 2”/“dtor 1”
    std::cout << sp->m_a;                      // “2”

    sp.reset();                                // “dtor 2”
    std::cout << sp->m_a;                      // Access violation!
```


Пример работы std::unique_ptr

```
void foo
{
    std::unique_ptr<CTrace> sp(new CTrace(1)); // “ctor 1”

    sp.reset(new CTrace(2));                  // “ctor 2”/“dtor 1”
    std::cout << sp->m_a;                      // “2”

    sp.reset();                               // “dtor 2”
    if (sp)
    {
        std::cout << sp->m_a;
    }

    // nothing
}
```

Пример работы std::unique_ptr

```
void foo
{
    std::unique_ptr<CTrace> sp1(new CTrace(1));

    std::unique_ptr<CTrace> sp2;
    sp2 = sp1;                                     // Illegal
}
```

Пример работы std::unique_ptr

```
void foo
{
    std::unique_ptr<CTrace> sp1(new CTrace(1));

    std::unique_ptr<CTrace> sp2;
    sp2 = sp1;                                // Illegal

    std::unique_ptr<CTrace> sp3(sp2);          // Illegal
}
```

Пример работы `std::unique_ptr`

```
void foo
{
    std::unique_ptr<CTrace> sp1(new CTrace(1));

    std::unique_ptr<CTrace> sp2;
    sp2 = std::move(sp1); // OK

    std::unique_ptr<CTrace> sp3(std::move(sp2)) // OK
}
```

Пример работы std::unique_ptr

```
std::unique_ptr<CTrace> SomeFunc(...)  
{  
    return std::unique_ptr<CTrace>(new CTrace(1));  
}
```

```
std::unique_ptr<CTrace> sp2;  
sp2 = SomeFunc(...);           // OK, move оператор присваивания
```



Ключевое слово delete

Явно помечает те методы, с которыми нельзя работать
(ошибка на этапе компиляции)

Ключевое слово delete

Явно помечает те методы, с которыми нельзя работать (ошибка на этапе компиляции)

```
class X
{
public:
    X() = default;
    X(const X& x) = delete;

};
```

```
int main()
{

    X a;
    X b(a);           // error

}
```


Ключевое слово delete

Явно помечает те методы, с которыми нельзя работать (ошибка на этапе компиляции)

```
class X
{
public:
    X() = default;
    X(const X& x) = delete;

    void bar(int) = delete;
    void bar(double) {...}

};
```

```
int main()
{

    X a;
    X b(a);           // error

    a.bar(2.4);
    a.bar(2);         // error

}
```

Ключевое слово delete

Явно помечает те методы, с которыми нельзя работать (ошибка на этапе компиляции)

```
class X
{
public:
    X() = default;
    X(const X& x) = delete;

    void bar(int) = delete;
    void bar(double) {...}

    void* operator new(std::size_t) = delete;
};
```

```
int main()
{

    X a;
    X b(a);           // error

    a.bar(2.4);
    a.bar(2);         // error

    X* c = new X;     // error
}
```

Реализация std::unique_ptr

```
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;

};
```

Реализация std::unique_ptr

```
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;
    }

};
```

Реализация std::unique_ptr

```
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)
    { ... }
};
```

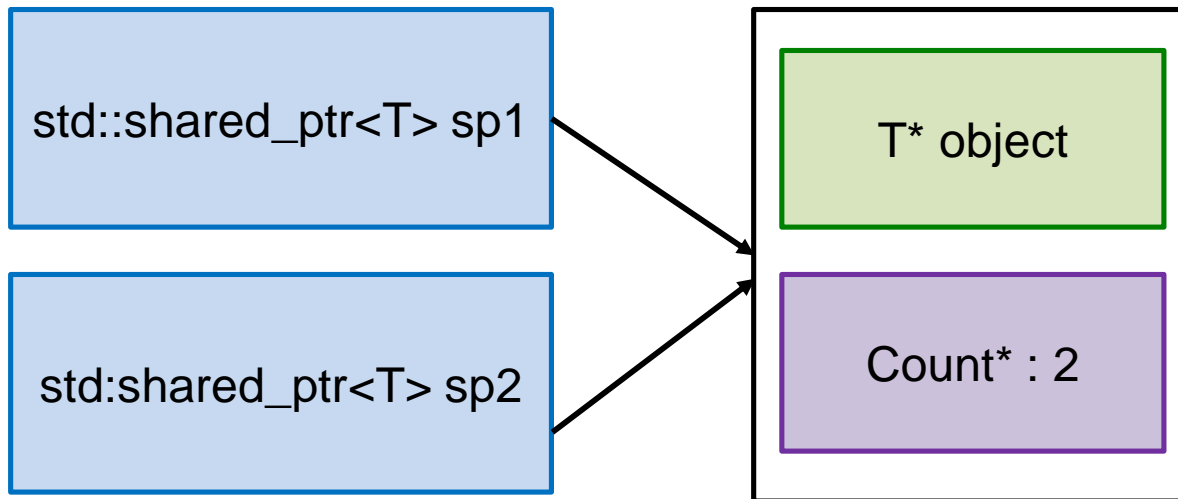


`std::shared_ptr`

Поддерживает совместное владение объектом

std::shared_ptr

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



Пример работы std::shared_ptr

```
std::shared_ptr<CTrace> f()                                     use_count
{
    std::shared_ptr<CTrace> sp1(new CTrace(1)); // “ctor 1”    1
```

Пример работы std::shared_ptr

	use_count
<pre>std::shared_ptr<CTrace> f() { std::shared_ptr<CTrace> sp1(new CTrace(1)); // “ctor 1”</pre>	1
<pre> std::shared_ptr<CTrace> sp2 = sp1; // sp1.get() == sp2.get()</pre>	2

Пример работы std::shared_ptr

	use_count
<code>std::shared_ptr<CTrace> f()</code>	
<code>{</code>	
<code> std::shared_ptr<CTrace> sp1(new CTrace(1)); // “ctor 1”</code>	1
<code> std::shared_ptr<CTrace> sp2 = sp1;</code>	2
<code> // sp1.get() == sp2.get()</code>	
<code> sp1.reset();</code>	1

Пример работы std::shared_ptr

	use_count
std::shared_ptr<CTrace> f() { std::shared_ptr<CTrace> sp1(new CTrace(1)); // “ctor 1”	1
std::shared_ptr<CTrace> sp2 = sp1; // sp1.get() == sp2.get()	2
sp1.reset();	1
return sp2; }	
void g() { std::shared_ptr<CTrace> sp3(f());	1

Пример работы std::shared_ptr

	use_count
std::shared_ptr<CTrace> f() { std::shared_ptr<CTrace> sp1(new CTrace(1)); // “ctor 1”	1
std::shared_ptr<CTrace> sp2 = sp1; // sp1.get() == sp2.get()	2
sp1.reset();	1
return sp2; }	
void g() { std::shared_ptr<CTrace> sp3(f());	1
} // “dtor 1”	0

std::shared_ptr: циклическая зависимость

```
class A
{
    std::shared_ptr<B> m_ptr;
};

class B
{
    std::shared_ptr<A> m_ptr;
}

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



`std::weak_ptr`

Решает проблему циклической зависимости

std::weak_ptr

- Решает проблему циклической зависимости
- **Не увеличивает** счетчик ссылок
- Получает доступ к тому же объекту, на который указывает std::shared_ptr (или другой std::weak_ptr), но не считается владельцем этого объекта
- Нельзя использовать напрямую (нет оператора ->)
- Может быть создан только из std::shared_ptr (или другого std::weak_ptr)

Пример работы std::weak_ptr

	use_count
std::shared_ptr<CTrace> sp1(new CTrace(1));	1
std::weak_ptr<CTrace> sp2(sp1);	1
...	
sp1.reset();	0
std::cout << sp2->m_n; // Illegal	

Пример работы std::weak_ptr

	use_count
std::shared_ptr<CTrace> sp1(new CTrace(1));	1
std::weak_ptr<CTrace> sp2(sp1);	1
...	
sp1.reset();	0
std::shared_ptr<CTrace> sp3 = sp2.lock();	
if (sp3)	
{	
std::cout << sp3->m_a;	
}	
else	
{	
std::cout << "Error";	
}	

std::weak_ptr

```
class A
{
    std::weak_ptr<B> m_ptr;
};

class B
{
    std::weak_ptr<A> m_ptr;
}

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

Советы по использованию умных указателей

1) Не используйте обычный указатель для хранения объектов

```
MyClass* a = new MyClass(...); // BAD PRACTICE!  
std::shared_ptr<MyClass> sp = new MyClass(...);
```

2) Не создавайте умные указатели из обычных указателей

```
int main()  
{  
    MyClass* a = new MyClass(...);  
    std::shared_ptr<MyClass> sp1(a);  
    std::shared_ptr<MyClass> sp2(a);  
    return 0;  
}
```

3) Не удаляйте обычные указатели, полученные из умных указателей

```
void foo()  
{  
    std::shared_ptr<MyClass> sp = new MyClass(...);  
    MyClass* a = sp.get();  
    delete a;  
}
```

std::make_shared() / std::make_unique()

```
void foo(std::shared_ptr<CTrace> a) { ... }

int main()
{
    foo(std::shared_ptr<CTrace>(new CTrace(1))); // usual way
    foo(std::make_shared<CTrace>(1));           // better way
}
```

- Изолируют не только операторы delete, но и new
- Выделяет память на счетчик одним блоком с объектом (std::shared_ptr)
- Для std::unique_ptr есть std::make_unique



Вопросы?

Что почитать?

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. https://en.cppreference.com/w/cpp/language/copy_elision
5. <https://habr.com/ru/post/470265/#RVO> - RVO и NRVO
6. <https://www.internalpointers.com/post/c-rvalue-references-and-move-semantics-beginners> - C++ rvalue references and move semantics for beginners
7. <https://habr.com/ru/post/348198/> - Понимание lvalue и rvalue в C и C++
8. <https://habr.com/ru/post/441742/> - Категории выражений в C++
9. <https://habr.com/ru/post/226229/> - Краткое введение в rvalue-ссылки
10. <http://stackoverflow.com/questions/3106110/what-are-move-semantics> - What is move semantics?
11. <http://msdn.microsoft.com/ru-ru/library/hh279676.aspx> - How to: Create and use unique_ptr instances
12. <http://habrahabr.ru/post/140222/> - Smart pointers для начинающих