returning by value

Typické implementace funkcí vracejících hodnotou

- Vracení nově zkonstruované hodnoty
  - S lokální proměnnou
    - [C++17] překladač musí
      - buďto provést *copy-elision* - proměnná tmp zanikne
      - nebo použít move-constructor v příkaze return - proměnná se nebude kopírovat

```
std::string concat( const std::string & a, const std::string & b)
{ auto tmp = a; tmp.append( b); return tmp; }
```

  - S anonymním objektem
    - [C++17] objekt vznikne až na finálním místě vně funkce

```
Complex add( const Complex & a, const Complex & b)
{ return Complex( a.re + b.re, a.im + b.im); }
```

    - V některých případech lze použít i zjednodušenou syntaxi

```
Complex add( const Complex & a, const Complex & b)
{ return { a.re + b.re, a.im + b.im}; }
```

Funkce vracející hodnotou

- Vracení struktur hodnotou – technické detaily

```
std::string concat( const std::string & a, const std::string & b)
{ auto tmp = a; tmp.append( b); return tmp; }
```

  - Použití výsledku v inicializaci:

```
void f() { std::string x = concat( y, z); use(x); }
```

  - Použití výsledku v přiřazení:

```
void g() { std::string x; x = concat( y, z); use(x); }
```

- Překlad před C++11 bez *copy-elision*
  - Ekvivalent v hypotetickém jazyce "C s metodami":

```
void concat( std::string * r, const std::string * a, const std::string * b)
{ std::string tmp; tmp.copy_ctor(a); tmp.append(b);
  r->copy_ctor(&tmp); tmp.dtor(); }
```

  - Překladač předpokládá, že volaná funkce volá konstruktor na objekt *r:

```
void f() { std::string x; concat( &x,&y,&z); use(x); x.dtor(); }
void g() { std::string x,t; x.ctor();
  concat(&t,&y,&z); x.copy_asgn(&t); t.dtor();
  use(x); x.dtor();}
```

  - Pomocný objekt t je nutný, protože x je již inicializován
  - ctor, copy_ctor, copy_asgn a dtor označují přeložené verze těchto C++ metod:

```
string();                                   // default-constructor
string(const string &);                     // copy-constructor
string & operator=(const string &);         // copy-assignment
~string();                                  // destructor
```

  - Pro string jsou tyto metody explicitně implementovány ve standardní knihovně

Funkce vracející hodnotou

- Vracení struktur hodnotou – technické detaily

```
std::string concat( const std::string & a, const std::string & b)
{ auto tmp = a; tmp.append( b); return tmp; }
```

  - Použití výsledku v inicializaci:

```
void f() { std::string x = concat( y, z); use(x); }
```

  - Použití výsledku v přiřazení:

```
void g() { std::string x; x = concat( y, z); use(x); }
```

- Překlad v C++11 bez *copy-elision*
  - Ekvivalent v hypotetickém jazyce "C s metodami":

```
void concat( std::string * r, const std::string * a, const std::string * b)
{ std::string tmp; tmp.copy_ctor(a); tmp.append(b);
  r->move_ctor(&tmp); tmp.dtor(); }
```

  - Překladač předpokládá, že volaná funkce volá konstruktor na objekt *r:

```
void f() { std::string x; concat( &x,&y,&z); use(x); x.dtor(); }
void g() { std::string x,t; x.ctor();
  concat(&t,&y,&z); x.move_asgn(&t); t.dtor();
  use(x); x.dtor();}
```

  - move_ctor a move_asgn jsou metody nově zavedené v C++11:

```
string(string &&);                  // move-constructor
string & operator=(string &&);     // move-assignment
```

  - jejich implementace u typu string je rychlejší než u copy-metod

Funkce vracející hodnotou

- Vracení struktur hodnotou – technické detaily

```
std::string concat( const std::string & a, const std::string & b)
{ auto tmp = a; tmp.append( b); return tmp; }
```

  - Použití výsledku v inicializaci:

```
void f() { std::string x = concat( y, z); use(x); }
```

  - Použití výsledku v přiřazení:

```
void g() { std::string x; x = concat( y, z); use(x); }
```

- Překlad s *copy-elision*
  - Proměnná tmp zaniká, místo ní překladač používá proměnnou x ve volající funkci
  - Ekvivalent v hypotetickém jazyce "C s metodami":

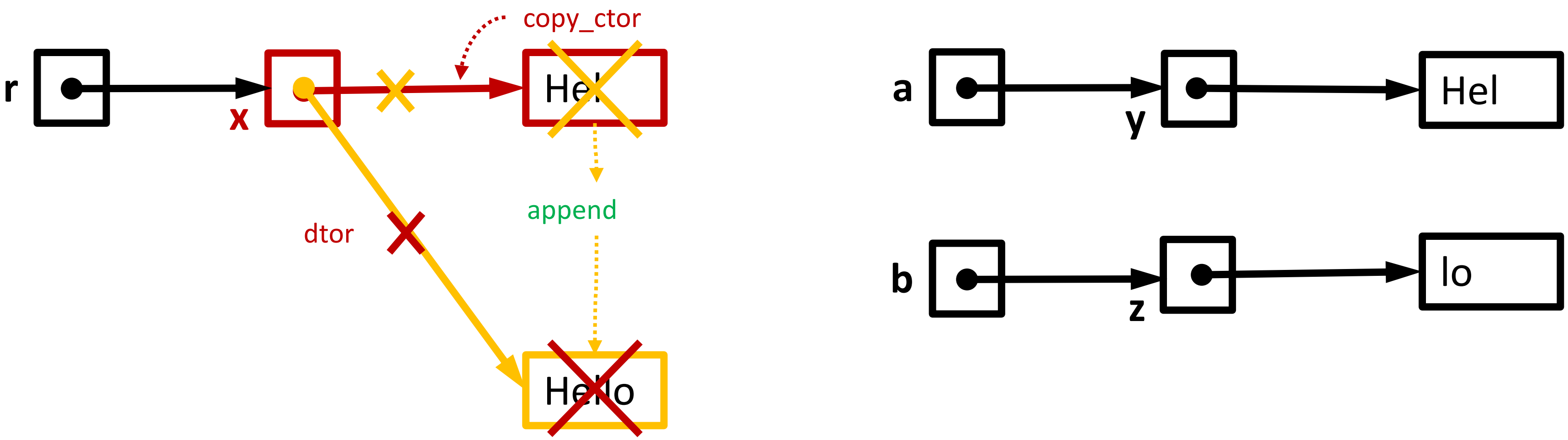
```
void concat( std::string * r, const std::string * a, const std::string * b)
{ r->copy_ctor(a); r->append(b); }
```

  - Překladač u volání o copy-elision neví, kód zůstává stejný:

```
void f() { std::string x; concat( &x,&y,&z); use(x); x.dtor(); }
void g() { std::string x,t; x.ctor();
  concat(&t,&y,&z); x.move_asgn(&t); t.dtor();
  use(x); x.dtor();}
```

- copy-elision mění pozorovatelné chování programu
  - explicitně povoleno normou C++

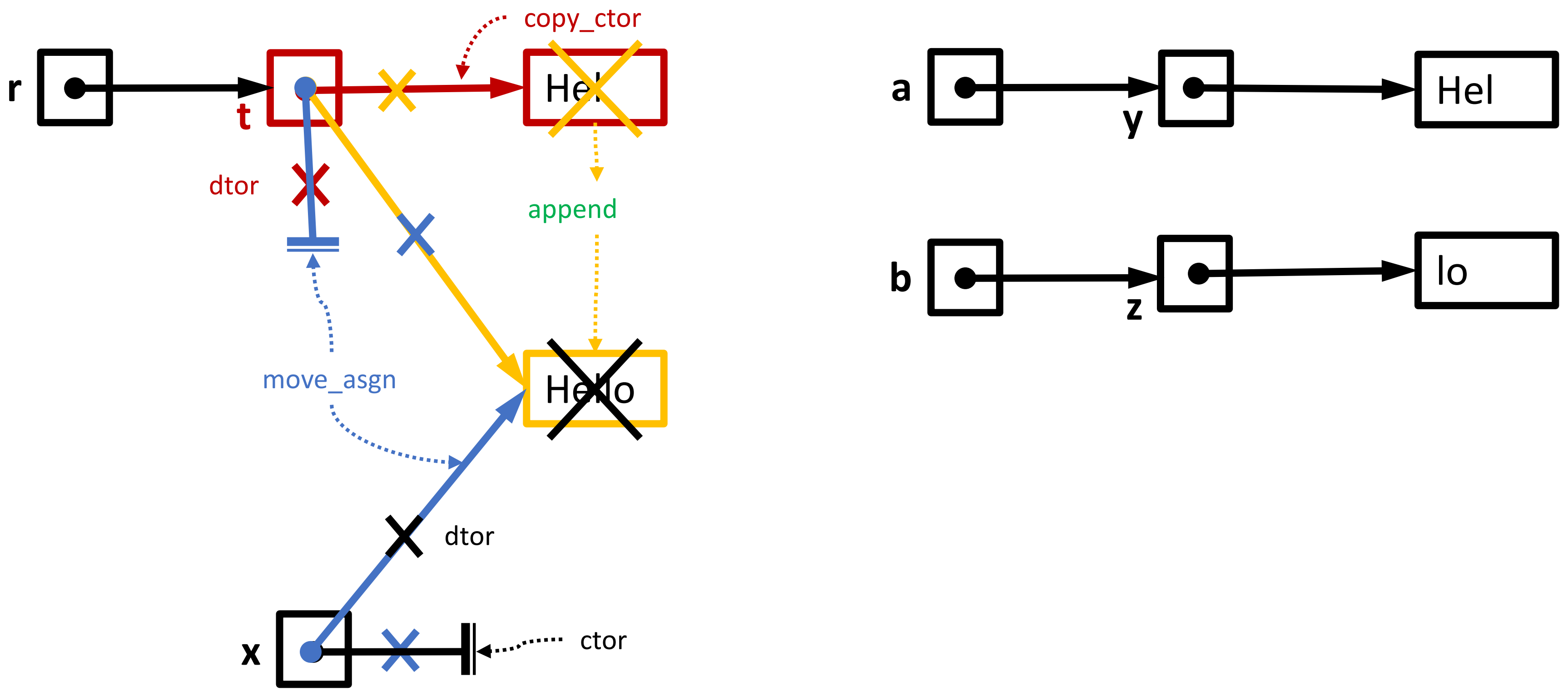
Funkce vracející hodnotou (v inicializaci)



- C-like equivalent

```
void concat( std::string * r, const std::string * a, const std::string * b)
{ r->copy_ctor(a); r->append(b); }
void f()
{ std::string x; concat( &x,&y,&z); use(x); x.dtor(); }
```

Funkce vracející hodnotou (v přiřazení)



- C-like equivalent

```
void concat( std::string * r, const std::string * a, const std::string * b)
{ r->copy_ctor(a); r->append(b); }
void g()
{ std::string x,t; x.ctor(); concat(&t,&y,&z); x.move_asgn(&t); t.dtor();
use(x); x.dtor();}
```

(Named) Return Value Optimization

- Norma jazyka nyní u funkcí, vracejících hodnotou, vyžaduje:
  - RVO: Povinné *move-elision*
    - Je-li výraz v příkaze return *prvalue* (stejného typu, jako návratový typ funkce)...

```
Complex operator+(/*...*/) { return Complex(/*...*/); }
std::string f(const std::string& a, const std::string& b) { return a+b; }
```

    - ... žádná pomocná proměnná nevzniká
      - objekt je zkonstruován přímo v místě, kde je očekávána návratová hodnota
      - volaná funkce/operátor vrací návratovou hodnotu přímo do tohoto místa
  - NRVO: Buďto *copy-elision* nebo implicitní *move*
    - Je-li výraz v příkaze return jméno lokální proměnné (ne parametru) ...
      - ... stejného typu jako návratový typ funkce ...
      - ... a všechny příkazy return v oblasti platnosti této proměnné jsou stejné ...

```
Complex operator+(/*...*/) { Complex v(/*...*/); /*...*/; return v; }
```

    - ... buďto je proměnná umístěna přímo tam, kde je očekávána návratová hodnota ...
      - ... příkaz return tedy fakticky zmizí ...
    - ... nebo je proměnná zachována, ale výraz v příkaze return je považován za r-value...
      - ... příkaz return tedy bude realizován move-constructorem
    - Poučení: U lokální proměnné v příkaze return nepište *move*

  - Podobná pravidla platí pro příkaz throw (bez ohledu na typ)

prvalue

- *prvalue* jsou umisťovány v pomocných proměnných, jen je-li to nutné
  - Pomocné proměnné nebudou vytvořeny pro *prvalue*...
    - ... v příkaze return – viz RVO
    - ... v inicializaci proměnné (stejného typu)

```
std::string x = std::string(/*...*/);  // same as std::string x(/*...*/);
auto y = std::string(/*...*/);
std::string z = x + y;                 // no temporary here
```

    - ... předáváné jako parametr hodnotou

```
void store(std::string p) { somewhere = std::move(p); }
store(x + y);   // operator+ stores the result directly into p
```

  - Pomocná proměnná je nutná, pokud je *prvalue* použita k inicializaci reference ...
    - ... při předávání parametrů odkazem

```
void inspect(const std::string & p) { std::cout << p; }
inspect(x + y); // operator+ stores the result into a temporary
```

      - Tyto pomocné proměnné mají životnost do konce příkazu
    - ... při inicializaci proměnných typu reference

```
const std::string & r = x + y;
auto && s = std::string(/*...*/);
const std::string & t = "Hello";    // implicit cast
```

      - Tato pomocná proměnná má stejnou životnost jako deklarovaná reference
      - Platí pouze pro vnější prvalue v inicializačním výrazu, vnitřní zanikají koncem deklarace!

```
auto && s = std::move(x + y);       // will CRASH!
```

    - Pro inicializaci referencí v návratových hodnotách se *prvalue* použít nedají!

Funkce vracející hodnotou

- Překladače nyní (povinně) implementují vracení hodnotou velmi efektivně
  - Lze efektivně realizovat i funkce vracející velké objekty

```
matrix operator*(const matrix& a, const matrix& b);
```

- Přehlednější použití ve srovnání s tradičním interface

```
void multiply(matrix &r, const matrix& a, const matrix& b);
void mm(int m, int k, int n, double* r, const double* a, const double* b);
```

- Funkce vracející hodnotou je možné chápat jako pojmenované konstruktory

```
class my_class {
public:
  my_class();
  my_class(/*some simple parameters*/);
  /*...*/
};
my_class create_my_class(/*some complex parameters*/)
{
  /* convert complex parameters to simple */
  return my_class(/*simple parameters*/);
}
auto var = create_my_class(/*...*/);
```

- Je možné vracet i nekopírovatelné objekty

```
std::ifstream open_input_file(/*...*/);
std::thread launch_computation(/*...*/);
```

Returning by value

- With move semantics and copy-elision, T pop_back() might be implemented
  - it still must return by value, but we can move the value quickly

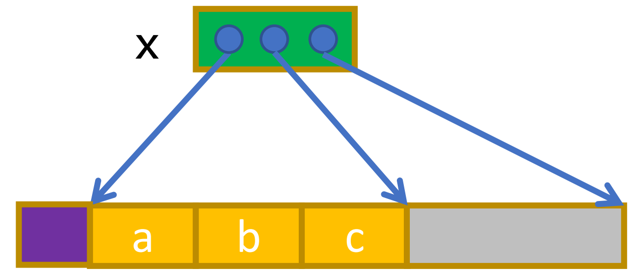
```
template< typename T> class vector_ng {
public:
  T pop_back()
  {
    T tmp = std::move(arr_[size_-1]);  // move-constructor
    arr_[size_-1].T::~T();             // destruct the object in the array
    --size_;                           // mark the object as unused
    return tmp;                        // copy-elision or move
  }
private:
  T * arr_;
  std::size_t size_;
};
```

- The speed relies on move-constructor and move-assignment of the type T
- Programmers shall equip their classes with fast move operations
  - If all the data members have fast move operations, the class will acquire it automatically
  - Otherwise, programmers need to implement the move (and copy) operations explicitly

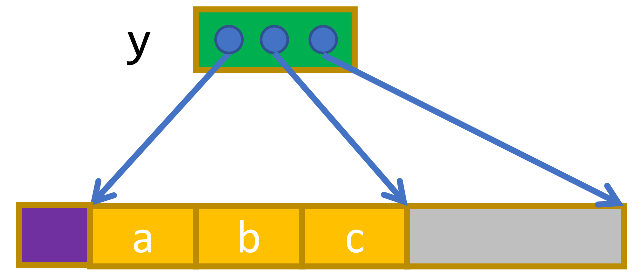
copy/move operations

Copy

```
std::vector< char> x { 'a', 'b', 'c' };
```



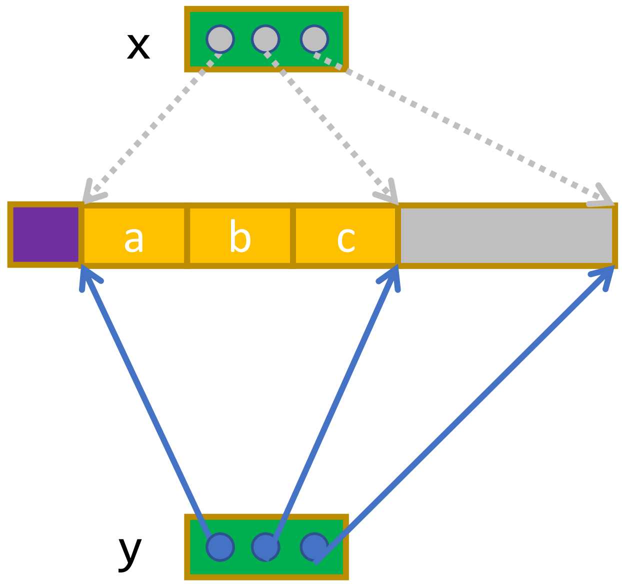

```
std::vector< char> y = x;
```



- A copy operation on containers and similar types
  - Requires allocation and copying of dynamically-allocated data
  - It is slow and may throw exceptions

Move

```
std::vector< char> x { 'a', 'b', 'c' };
```




```
std::vector< char> y = std::move(x);
```

- After moving, the source is *empty*
  - Exact meaning depends on the type
- A move operation usually does no allocation
  - It is fast and does not throw exceptions

- Move operation is invoked instead of copy, if
  - the source is explicitly marked with std::move(), or
  - the source is an r-value
    - temporary object, which cannot be accessed repeatedly
      - return values from functions which return by value
      - explicitly created temporary objects
      - results of casts etc.

  - std::move
    - actually a cast from lvalue-reference to rvalue-reference

```
template< typename T>
T && move(T & x) { return static_cast<T &&>(x); }   // simplified
```

    - std::move does NOT move anything
    - the cast (usually) changes the behavior AFTER the std::move call

```
T z = y;              // invokes T(const T&) because y is an l-value
T z = std::move(y);   // invokes T(T&&) because std::move(y) is an r-value
```

- The meaning of copy and move operations depends on the type
  - The behavior is implemented as four special member functions
    - copy-constructor – called when initializing a new object by copying

```
T( const T &);
```

    - move-constructor – called when initializing a new object by moving

```
T( T &&);
```

    - copy-assignment – called when copying a new value to an old object

```
T & operator=( const T &);
```

    - move-assignment – called when moving a new value to an old object

```
T & operator=( T &&);
```

  - if not implemented by the programmer, the compiler will create them
    - only if some (rather complex) conditions ensuring backward compatibility are met
      - otherwise the respective copy/move operations are not supported by the type
    - the compiler-generated implementation calls the corresponding functions for all data members (and base classes)
      - if you follow C++11 guidelines, this behavior will probably meet your needs
  - for elementary types (numbers, T *), move is implemented as copy
    - it may cause inconsistency between number and container members
  - when containers are moved, all elements are also moved
    - the source container becomes empty (except std::array which cannot be resized)

- Consider what happens when your class is going to die...
- ... can all the data members clean-up themselves?
  - Numbers need no clean-up
  - Smart pointers will automatically clean up their memory blocks if necessary
  - Raw (T*) pointers will just disappear, they can not do any clean-up automatically
    - If they are just observers, it is O.K. - they are not responsible for cleaning
    - If they represent ownership, you will need to call delete in a destructor

```
class T { public:
  T(/*...*/) : p_( new U(/*...*/)) {}  // plain old dynamic allocation
  ~T() { delete p_; }  // destructor required
private:
  U * p_;       // owner of a memory block
};
```

- Now, what happens if you copy the owner class T bit-by-bit?
  - There will be two T objects containing pointers to the same object U
    - The second call to ~T() will cause CRASH due to double delete on the same object
    - It is impossible to determine that an object was already deleted
  - Instead of shallow copying, deep copy must be used for T

- The Rule of Five:

- If something forced you to write the destructor,
  you also have to write the four copy/move functions

  - The implementation of the four by the compiler would not fit your needs
    - Your destructor is unlikely to survive double invocation on shallow copies
    - Besides ownership pointers, it also applies to open files, locks, ...
  - You can also disable them if you don't need copyable/movable class:

```
T( const T &) = delete;
T( T &&) = delete;
T & operator=( const T &) = delete;
T & operator=( T &&) = delete;
```

- Implementing the Five functions is demanding and error-prone
  - Avoid using U* pointers where ownership is required
  - Use only types that can take care of themselves

The Rule of Five – possible scenarios

- All elements support copy and move in the required fashion
  - None of the Five methods required
  - Beware of the incoherence between numbers and smarter elements:

```
class matrix { private: std::vector<float> v_; std::size_t rows_, cols_; };
```

    - Move makes the source vector empty but rows_/cols_ remain nonzero!
    - You may need explicit implementation of move and default copy
- All elements support copy and move but copying has no sense
  - Living objects in simulations/games etc.
  - Disable copy methods by "= delete"
  - If move methods remain useful, they have to be made accessible by "= default"
    - Touching any of the four methods automatically disables the others (C++20)
- Elements support move in the required fashion, but copying is required
  - Copying elements does not work or behaves differently than required
    - E.g., elements are unique/shared_ptr but the class requires deep copy semantics
  - Implement copy methods, enable move methods by "= default"
- Elements do not support copy/move in the required way
  - Implement all the copy and move methods and the destructor

Virtual destructor

- Classes at the root of an inheritance hierarchy (usually abstract classes) must have a virtual destructor:

```
class C { virtual ~C() {} };
```

  - It enforces an advanced implementation of delete for pointers to the class
    - For speed, the default implementation of delete is dumb
  - A typical use of inheritance:

```
class D : public C { std::shared_ptr<Z> zp; }
```

    - A derived class object is dynamically allocated

```
D * dptr = new D;
```

    - A pointer to the derived object is then assigned to a pointer to the base class
      - This assignment is the core motivation for inheritance

```
C * cptr = dptr; // implicit conversion "derived to base class pointer"
```

    - Finally, the object is destroyed using the pointer to the base class
      - The compiler does not know the type of the object being deleted!

```
delete cptr;    // if C::~C() is virtual, it deletes the complete D object
```

    - Without virtual destructor, data members of derived classes will remain undestructed!
      - With multiple inheritance, the delete will also damage the allocation mechanism!
  - The same problem applies to smart pointers
    - Destructor of a smart pointer invokes delete on a raw pointer

Abstract classes

- Classes at the root of an inheritance hierarchy (usually abstract classes) must have a virtual destructor:

```
class AbstractClass { virtual ~AbstractClass() {} };
```

- Such classes are usually used solely as dynamically allocated objects
  - std::vector<AbstractClass> is a NONSENSE in C++
    - Such a container cannot store any derived class!
  - std::vector<std::unique_ptr<AbstractClass>> is the correct solution
- With dynamically allocated objects, move is usually not needed
  - The (smart) pointers to them are moved instead
- Often, objects with inheritance also have some kind of identity
  - Copying such objects usually has no sense
- It is a good idea to disable copy and move methods for abstract classes
  - The disablement will automatically propagate to derived classes
  - Sometimes, a destructor is needed to clean-up a derived class
    - The disablement makes the rule-of-five satisfied
- There are two completely different worlds inside C++
  - Classes used as values – no inheritance, passed by reference, returned by value
  - Classes with inheritance – dynamically allocated, immovable, passed as pointers