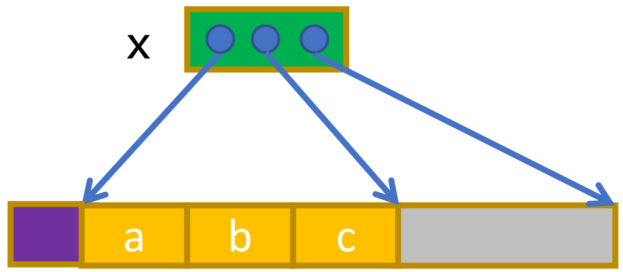
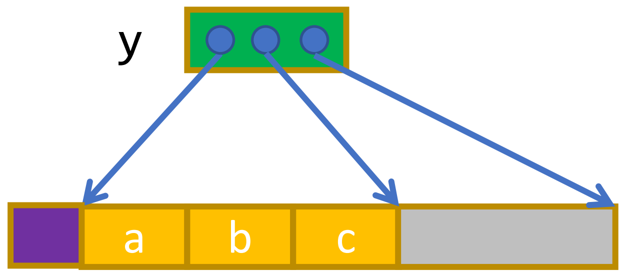
copy/move operations

Copy

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



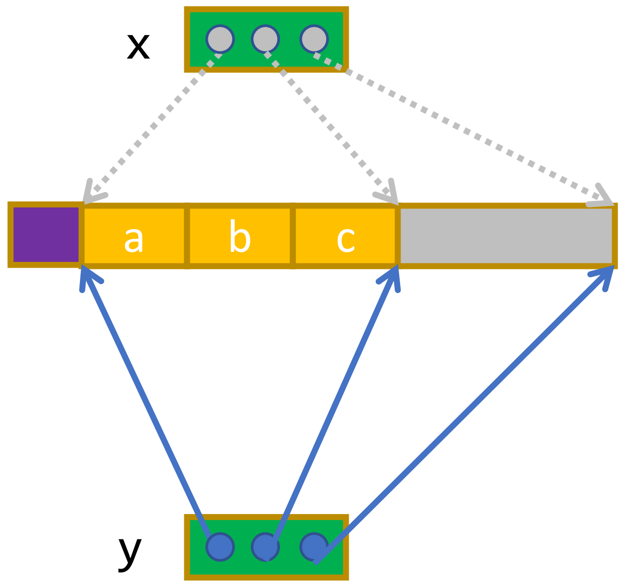

```
std::vector< char> y = x;
```



- 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

- 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() { delete p_; }  // destructor required
  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

Dynamic allocation

- Use smart pointers instead of raw (T *) pointers

```
#include <memory>
```

  - one owner (pointer cannot be copied)
    - negligible runtime cost (almost the same as T *)

```
void f() {
  std::unique_ptr< T> p = std::make_unique< T>();   // invokes new
  std::unique_ptr< T> q = std::move( p);  // pointer moved to q
  // p is nullptr now
}
```

  - shared ownership
    - runtime cost of reference counting

```
void f() {
  std::shared_ptr< T> p = std::make_shared< T>();   // invokes new
  std::shared_ptr< T> q = p; // pointer copied; object shared between q and p
}
```

- Memory is deallocated when the last owner disappears
  - Destructor of (or assignment to) the smart pointer invokes delete when required
  - Reference counting cannot deallocate cyclic structures

Dynamic allocation in C++11

- unique_ptr is uncopiable, shared_ptr is expensive to copy
  - avoid copying whenever possible

- When passing ownership, the parameter of the receiving function may be
  - passed by value

```
void store_pointer(std::shared_ptr<T> a) {
  storage_ = std::move(a);
}
```

  - passed by r-value reference

```
void store_pointer(std::shared_ptr<T> && a) {
```

    - the ownership transfer may be conditional

```
  if ( /*...*/ )
    storage_ = std::move(a);
}
```

- In **both** cases, pass the actual argument using **move**:

```
store_pointer(std::move(p));
```

  - if passed by value, the ownership is immediately moved to the argument a
    - and later moved again to the storage
  - if passed by reference, the ownership is moved directly to the storage
    - and may remain in the actual argument if not actually moved
    - if the calling function wants to use p after calling store_pointer(std::move(p)), there must be a mechanism informing it whether store_pointer actually moved or not

- unique_ptr is uncopiable, shared_ptr is expensive to copy
  - avoid copying whenever possible

- If you don't need to pass ownership, do not pass smart pointers
  - Use a raw pointer - **T *** or **const T ***
    - in this case, it is termed a *(modifying) observer* (to distinguish from old-style owning T *)
  - Raw pointers are always passed by value

```
void store_pointer(T * a) {
  storage_ = a;
}
```

- If the actual argument is a smart pointer, it must be explicitly converted

```
std::shared_ptr<T> p = /*...*/
store_pointer(p.get());
store_pointer(&*p);
```

  - The &* version is preferred – it works also on iterators or raw pointers
    - It is actually a user-defined operator***** followed by the built-in **&**

- The observers are not considered co-owners
  - The object may be destructed by an owner with observers present
  - It is the programmers responsibility to avoid using observers after owners die
    - This is the reason why the smart-to-observer conversion is not implicit

Using pointers in modern C++

- Owner of object
  - std::unique_ptr< T>, std::shared_ptr< T>
  - Use only if objects must be allocated one-by-one
    - Possible reasons: Inheritance, irregular life range, graph-like structure, singleton
    - For holding multiple objects of the same type, use std::vector< T>
  - std::weak_ptr< T>
    - To enable circular references with std::shared_ptr< T>, used rarely
- Modifying observer
  - T *
    - In modern C++, native (raw, T*) pointers shall not represent ownership
  - Save T * in another object which needs to modify the T object
    - Beware of lifetime: The observer must stop observing before the owner dies
    - If you are not able to prevent premature owner death, you need shared ownership
- Read-only observer
  - const T *
  - Save const T * in another object which needs to read the T object
- Besides pointers, C++ has references (T &, const T &, T &&)
  - Used (by convention) for temporary access during a function call etc.

Owners and observers

- Example – unique ownership

```
auto owner = std::make_unique< T>();      // std::unique_ptr< T>
```

  - Observer

```
auto modifying_observer = owner.get(); // T *
auto modifying_observer2 = &*owner;    // same effect as .get()
```

  - Read-only observer

```
const T * read_only_observer = owner.get();  // implicit conversion of T * to
const T *
auto read_only_observer2 = (const T *)owner.get();  // explicit conversion
const T * read_only_observer3 = modifying_observer; // implicit conversion
```

- Owner pointers can point only to a complete dynamically allocated block
  - Or to a base object (with virtual destructor) from which the complete object is derived
- Observer pointers can point to any piece of data anywhere
  - Parts of objects

```
auto part_observer = & owner->member;
```

  - Static data

```
static T static_data[ 2];
T* observer_of_static = & static_data[ 0];
```

  - Local data (beware: their lifetime is limited – avoid propagating observers outside of their scope)

```
void g( T * p); // note: reference T& instead of pointer is preferred here
void f() { T local_data; g( & local_data); }
```

- Dynamic allocation is slow
  - compared to static/automatic storage
  - the reason is cache behavior, not only the allocation itself
- Use dynamic allocation only when necessary
  - variable-sized or large arrays
    - in most of these cases, dynamic allocation is used indirectly through containers
  - polymorphic containers (containing various objects using inheritance)

```
std::vector<std::unique_ptr<common_base_class>>
```

  - object lifetimes not corresponding to function invocations
    - however, this case can often be solved by moving the object contents
- For speed, avoid data structures with individually allocated items
  - linked lists, binary trees, ...
    - std::list, std::map, ...
  - prefer contiguous structures (vectors, hash tables, B-trees, etc.)
  - avoiding is difficult - do it only if speed is important

- This is how C++ programs may be made faster than C#/java
  - C#/java requires dynamic allocation of every class instance