Effective C++ Programming

NIE-EPC (v. 2021):

NEW[] AND DELETE[], OBJECT CONTENT, VECTOR CLASS

String — reallocation

- Adding content (characters) to `String` owner class:
  - If the *actual size + size of the added content* is greater than the *allocated capacity*, reallocation must take place.

```
class String {  // without SSO for sake of simplicty
  size_t capacity_, size_;
  char* data_;
  ...
public:
  void push_back(char c) {
    if (size_ == capacity_)
      reserve( capacity_ ? 2 * capacity_ : 1 );  // for example, double the capacity
    data_[size_++] = c;  data_[size_] = '\0';    // add c at the end
  } ...
```

- Reallocation needs to:
  1) allocate new buffer with increased capacity,
  2) copy the owned string of characters from the old buffer to the new,
  3) deallocate the original buffer.

```
  void reserve(size_t capacity) {
    if (capacity <= capacity_) return;    // reserver only increases capacity
    char* data = new char[capacity + 1];  // ad 1)
    memcpy(data, data_, size_ + 1);       // ad 2)
    delete[] data_;                       // ad 3)
    data_ = data;  capacity_ = capacity;
  }
```

String — reallocation *(cont.)*

- We used the *"array" form of new-expression* (new[ ]) here, which "binds" *storage allocation* (for multiple objects) with their *initialization*.
- Similarly, `delete[]` "binds" *destruction* of all objects with *storage deallocation*.
- *Alternatively*, we can detach both:

```
void reserve(size_t capacity) {
  if (capacity <= capacity_) return;
  char* data = (char*)::operator new(capacity + 1);  // allocate storage only
  for (size_t i = 0; i <= capacity_; i++)
    new (data + i) char;                              // initialize objects/elements
  memcpy(data, data_, size_ + 1);
  ::operator delete(data_;)                           // deallocate original storage
  data_ = data; capacity_ = capacity;
}
```

- But there would be effectively *no difference*:
  - *Default initialization* for char object is effectively no-op (has no observable behavior; [link]).
  - There is no destructor for char ⇒ destruction does not require any action.

new[] *vs* new[]() *vs* new[]{}

- If we used…

```
for (size_t i = 0; i < n; i++) new (data + i) char();  // note that trailing '()'
```

- …or…

```
for (size_t i = 0; i < n; i++) new (data + i) char{};  // note that trailing '{}'
```

- …instead of…

```
for (size_t i = 0; i < n; i++) new (data + i) char;
```

- …all the char objects would be initialized to value `'\0'`.
  - In our `String::reserve()` case, this makes no sense, since elements are then overwritten by `memcpy` anyway (it would be a *wasted overhead*).
- The same holds for *new-expression*:

```
char* s1 = new char[n];    // all elements have unspecified values
char* s2 = new char[n]();  // all elements have values '\0'
```

- For class types, there is no difference:
  - All the elements are initialized by default constructor in all 3 cases.

String *vs* vector

- *String class* = owner of a string of characters.
  - String of characters = dynamic array of characters.
- *Vector class* = owner of a dynamic array of elements of any type.
  - ⇒ Vector needs to be a *class template*.
  - Type of owned objects is a template parameter.
  - It is called a "*value type*".
- Custom *vector-class* design:

```
template <typename T>
class Vector {
  size_t capacity_, size_;
  T* data_;
  ...
public:
  using value_type = T;
  ...
```

- *Note* — vector guarantees that its elements are placed in a contiguous storage ⇒ fast element access.

```
T* data() { return data_; }
T& operator[](size_t n) { return *(data_ + n); }
...
```

Vector — reallocation — *problems*

- Will the reallocation work the same way as for `String`?

```
template <typename T> class Vector {
  ...
  void reserve(size_t capacity) {
    if (capacity <= capacity_) return;
    T* data = new T[capacity];                  // new expression (array form)
    memcpy(data, data_, size_ * sizeof(T));     // copy content
    delete[] data_;                             // delete expression (array form)
    data_ = data; capacity_ = capacity;
  }
```

- Problems with this approach:
  - *First*, it wouldn't support types without default constructor:

```
struct Y {
  Y(int);  // suppresses generation of...
  ~Y();    // ...default Y() constructor
};
```

```
Vector<Y> v;
v.reserve(10);  // error: no matching...
                // ...function for call...
                // ...to 'Y::Y()'
```

- This is not "conforming" to `std::vector`, which does not require a *default-constructible* value type.

```
std::vector<Y> v;
v.reserve(10);  // OK :-)
```

Vector — reallocation — *problems (cont.)*

- *Second*, such an approach would be inefficient.

```
struct X {
  X();  // default constructor
  ~X();
};
```

```
std::vector<X> v;
v.reserve(10);
// no owned elements, only reserved storage
```

- *Required effect:*
  - v is empty ⇒ it does not own any elements.
  - It has reserved storage for 10 elements ⇒ when adding up to 10 elements, no reallocation is required.
- With `new[]` *expression*:
  - 10 elements would need to be *default-constructed* and vector would need to *take care of them* (own them) ⇒ it would be inefficient.
  - It wouldn't be conforming with respect to how vectors are required to work (*for instance*, `reserve` on empty vector must have $O(1)$ time complexity).
- *Third*, `memcpy` cannot be used to copy binary representation of any data type (*more on this later*):

```
memcpy(data, data_, size_ * sizeof(T));  // works only for some T
```

Vector — reallocation — *storage allocation*

- *Solution* = detaching *storage allocation* from *object initialization*...

```
template <typename T> class Vector {
  ...
  void reserve(size_t capacity) {
    if (capacity <= capacity_) return;
    T* data = (T*)::operator new(capacity * sizeof(T));  // storage allocation only
    ...                                                   // initialize objects (see later)
  }
```

- ...and *object destruction* from *storage deallocation*:

```
  void clear() {
    for ( ; size_ > 0; size_--)
      (data_ + size_ - 1)->~T();  // destructs all owned objects/elements
  }

  ~Vector() {
    clear();
    ::operator delete(data_);     // storage deallocation
  }
```

- *To-be-solved:*
  - How to get elements from old to new storage during reallocation?
  - And, then, how to add elements into vector?

Vector — reallocation — “*moving*” *elements*

```
void reserve(size_t capacity) {
  if (capacity <= capacity_) return;
  T* data = (T*)::operator new(capacity * sizeof(T));
  ... // get elements from old (data_) to new (data) storage - HOW???
```

- We would now need to “move” the vector elements from the old storage to the new one.
- Is it possible to move an object from one storage to another one during its lifetime?
- **NO** — C++ does not support such operation.
  - Storage of an existing object is guaranteed to be persistent/same during its entire lifetime.
  - There is no such thing as moving the objects between storage (such as between different memory locations) in C++.
  - Quote from the C++ Standard draft version:
    “*An object occupies a region of storage in its period of construction, throughout its lifetime, and in its period of destruction.*” [intro.object/1/3]

Vector — reallocation — *"moving"* elements

- New storage must contain objects/elements after reallocation.
- The only way how to create them is to *initialize/construct* them.
- Original storage contains `size_` elements.
- In reallocation, only storage capacity in increased; no elements are added or removed.
- ⇒ New storage must also contain `size_` elements.
- These elements need to be initialized there:

```
void reserve(size_t capacity) {
  if (capacity <= capacity_) return;
  T* data = (T*)::operator new(capacity * sizeof(T));  // new storage allocation

  for (size_t i = 0; i < size_; i++)
    new (data + i) T( ??? );  // initialization of size_ elements in new storage
```

- How to initialize them?
  - We want the new elements to be *"equal/same"* as the original elements.
- *Same/equal* objects = objects having *same/equal* content.
- What is a *"content" of an object?*

Object content

- *"(Logical) content"* of an object — content given by the semantics of object type.
  - Basically, *semantics of a type* defines which ("real-world") entity represent objects of this type in a C++ program.
  - For instance, semantics of type `int` = its objects represent integer numbers from some range (each object logically "contains" such a number).
- *Illustrative example:*

```
int* pi { new int(1) };
std::unique_ptr<int> upi { new int(1) };
```

- "*Raw*" pointer `pi`:
  - Raw pointers do not "own" objects, they only point to them.
  - *Logical content* = address of the allocated object.
- "*Smart*" pointer `upi`:
  - Smart pointers do "own" allocated objects.
  - *Logical content* = ownership/management of the allocated object.

Object content and binary representation

- *Another example:*

```
std::unique_ptr<int> upi { new int(1) };
int* pi = upi.get();
```

- Both *raw pointer* `pi` and *unique pointer* `upi` now refer/point to the same object.
- What is their *binary representation* (byte content of their storage)?
- It is the very same:
  - We can look at it, for example, in a *debugger*, or inspect it even in a *program:* https://godbolt.org/z/1Kn95W1jr.
  - Exemplary observed binary rep.: 64-bit address `0x000000B0DE710000`.
- *Reason* — simplified *unique-pointer class*:

```
template <typename T>
class unique_ptr {
  T* ptr_;
public:
  ...
};
```

- ⟹ It just "wraps" an ordinary raw pointer.

Object content and binary represent. *(cont.)*

- The same binary representation of objects generally represents different content given by the semantics of their types.
- In relation to *objects binary representation*, there are two cases:
- *Case I.* — *logical content* is fully given by object *binary representation*.
- *Example I:*
  - Logical content of object `i` of type `int` is an *integer number*.
  - This number is "*encoded*" in binary representation of `i`.

```
int i = -1;  // 32-bit binary representation 0xFFFFFFFF represents int value -1
```

- *Example II:*
  - Logical content of object `pi` of type `int*` is an *address* (of some object of type `int`).
  - This address is "encoded" in binary representation of `pi`.

```
int* pi = new int(1);  // 64-bit binary rep., e.g., 0x00000000021D9EB0 represents address
```

Object content and binary represent. *(cont.)*

- *Case II.* — object's *logical content* is *"more"* than just its *binary representation*.
- *Example — smart pointers*:

```
std::unique_ptr<int> upi { new int(1) };
```

  - Logical content of an object `upi` of type `unique_ptr<int>` is an ownership of a dynamically-allocated object of type `int`.
- Logical content here may have much more generic meaning than just ownership of other objects.
- *Examples:*

```
std::fstream f("test.bin", std::ios::binary);
std::thread t( []{ std::cout << "Hello world from thread t!"); } );
```

  - Logical content of object `f` of type `std::fstream` is an open file stream (ownership of file handler and the corresponding management).
  - Logical content of object `t` of type `std::thread` is a running thread of execution (*ditto*).

Object content-related operations

- *Special operations* related to object content:
  - *Content copying* — destination object has the same content as the source object, while the content of the source object is preserved.
  - *Content moving* — destination object has the same content as the source object, while the content of the source object may not be preserved.
  - *Content destruction (release)* — object has no content after destruction.
- For ***Case I.* types**:
  - *Copying* content of objects = copying their binary representation.

```
int i = 1, j;
memcpy(&j, &i, sizeof(int));  // j has same conent as i (represents number 1)
```

    - It even generates the same assembly as assignment `j=i`: [*Godbolt link*].
  - *Moving* content of objects = there is no such thing.
    - We can copy bytes, but there is no "moving of bytes" (zero bytes are not any special).
  - *Destruction* of content = no operation/action.
    - What would mean to destruct an integer number?

Object content-related operations *(cont.)*

- For ***Case II.* types**:
  - These special content-related operations involve more than just working with binary objects representations.
  - $\Rightarrow$ They must be either customized (*custom-defined*) or disabled.
  - These operations have the form of:
    - *Copy constructors* (1) and *copy assignment operators* (2) for content *copying*.
    - *Move constructors* (3) and *move assignment operators* (4) for content *moving*.
    - *Destructors* (5) for content *destruction/release*.
- *The rule of 3/5*:
  - Usually, once any of these operations needs to be customized/disabled, all of them should be as well.
  - If *content moving* does not make sense (according to the type semantics), moving may be "*merged*" with copying ($\Rightarrow$ only 3 operations are then needed).
  - *Note* — before C++11, there was no *content moving* ($\Rightarrow$ rule of 3).

Trivial copyability

- ***Case I.*** **types** (and their objects) are called to be "***trivially-copyable***":
  - *Content copying* = (*trivial*) copying of binary representation.
  - *Content moving* does not exist.
  - *Content destruction* = effectively no-op (no action required).
- **Trivially-copyable types:**
  1) All non-class types:
     - *character* types, *integer* types, *floating-point* types, *pointers*, `bool`, *enums*).
  2) Classes where:
     - none of the 5 special operations is custom-defined or disabled (deleted);
     - and, the same holds for types of all class sub-objects (*non-static member variables* and *base classes*).
  3) Arrays of trivially-copyable objects.
- Content of objects of *trivially-copyable* types may be, for example, set by `memcpy`.

Non-trivial copyability

- Once any of the special copy/move/destruction operations needs to be customized/disabled, the type is not trivially-copyable — ***Case II.* types**.
- *Example — unique pointers*:
  - *Content* = *ownership* of dynamically-allocated object.
  - ⇒ Destruction of *unique pointer* = destruction of owned object + deallocation of its storage:

```
template <typename T>
class unique_ptr {
  T* ptr_;
public:
  T(T* ptr) : ptr_(ptr) { }
  ~T() { delete ptr_; }  // content destruction
  ...
};
```

- ⇒ *unique pointers* are not *trivially-copyable*.
- *General rule* — binary representation of objects of *non-trivially-copyable types* may not be set/updated directly.
  - Such operations result in *undefined behavior (UB)*.
  - *For instance*, content of non-trivially-copyable objects cannot be set by `memcpy`.

Trivial vs non-trivial copyability

- *Raw pointers* are *trivially-copyable*:

```
int* pi { new int(1) };
int* pj;
memcpy(&pj, &pi, sizeof(int*));
// pj is guaranteed to have same content (address of allocated object) as pi
```

  - Content of `pi` is an address of an object of type `int`.
  - Copying its binary representation into `pj` makes `pj` to have the very same content as `pi` — the address of the same object.
- However, the same does not hold for *unique pointers*:

```
std::unique_ptr<int> upi { new int(1) };
std::unique_ptr<int> upj;
memcpy(&upj, &upi, sizeof(std::unique_ptr<int>));  // wrong/illegal !!!
```

- What can get wrong in this case (technically, it's *UB*)?
  - Content of a *unique pointer* is ownership that is unique.
  - Copying binary representation effectively makes a single object owned by two unique pointers (`upi` and `upj`).
  - This breaks the semantics of unique pointers, may lead to double delete, etc…

Non-trivial copyability — *unique pointers*

```
~T() { delete ptr_; }  // content destruction
```

- We have defined *content-release/destruction* operation for *unique pointer class*.
- What about other *special content-related* operations?
- *Content copying*:
  - Content of a *unique pointer* is ownership of an object.
  - ⇒ Copying content = copying ownership of the owned object.
  - However, with *unique pointers*, owned object can be owned only by a single *unique pointer* ("*unique ownership*").
  - ⇒ Content of *unique pointers* (*ownership*) **cannot be copied**.
  - ⇒ Corresponding *content copying* operations need to be disabled:

```
T(const T&) = delete;             // disabled (deleted) copy constructor
T& operator=(const T&) = delete;  // disabled (deleted) copy assignment operators
```

Non-trivial copyability — *unique ptrs (cont.)*

- *Content moving*:
  - ⇒ Moving content = *moving ownership* of the owned object from *source* to *destination unique pointer*.
  - Before moving, the object was owned by the *source unique pointer*.
  - After moving, it will be owned by the *destination unique pointer*.
  - Unique ownership ⇒ they cannot own the object both at once.
  - ⇒ The ownership of the object by the *source unique pointer* — its content — needs to be *destructed/released*.
  - ⇒ We need to get the source unique pointer into a state where it does not own any object ("*empty*" *unique pointer*).
    - This state will be indicated by `ptr_` having `nullptr` value.

```
T(T&& src) : ptr_(src.ptr_) { str.ptr_ = nullptr; }  // move constructor

T& operator=(T&& src) {                              // move assignment operator
  if (this != &src) {    // check for self-assignment
    delete ptr_;         // get rid of current content
    ptr_ = src.ptr_;     // acquire ownership for myself
    src._ptr = nullptr;  // release ownership for source (to avoid double-ownership)
  }
  return *this;
}
```

Non-trivial copyability — *unique ptrs. (cont.)*

- More-complete simple *unique-pointer* class code:

```
template <typename T>
class unique_ptr {
  T* ptr_;

public:
  T() : ptr_(nullptr) { }    // default constructor: empty state = ownership of no object
  T(T* ptr) : ptr_(ptr) { }  // converting constructor: acquire ownership of provided object

  T(const T&) = delete;             // disabled (deleted) copy constructor
  T& operator=(const T&) = delete;  // disabled (deleted) copy assignment operators

  T(T&& src) : ptr_(src.ptr_) { str.ptr_ = nullptr; }  // move constructor

  T& operator=(T&& src) {                              // move assignment operator
    if (this != &src) {
      delete ptr_;
      ptr_ = src.ptr_;
      src._ptr = nullptr;
    }
    return *this;
  }

 ~T() { delete ptr_; }  // destructor

  T* get() { return ptr_; }         // get raw pointer to the owned object
  T& operator*() { return *ptr_; }  // get access (by reference) to the owned object
  ...
};
```

- *Note* — `std::unique_ptr` is more complicated.
  - It provides more member functions, *custom deleters* support, etc…

Non-copyable types

- *Note* — *copy operations* are *deleted implicitly* once *move operations* are *custom-defined*.
- Types (and their objects) *without copy semantics* — having disabled/deleted content copying operations — are called "non-copyable".
- Some other examples of library *non-copyable* types:
  - *File streams* — `std::fstream`, `std::ifstream`, `std::ofstream`:
    - *Content* = open file (ownership of *open file stream/handler*).
    - Logically, open file cannot be managed by multiple owners.
    - How would, for example, be resolved writes from multiple file stream objects?
  - *Thread "handlers"/owners* — `std::thread`:
    - *Content* = running thread of execution (its ownership/management).
    - Running thread may be managed only by a single `std::thread` owner.
  - *Mutex owner* — `std::mutex`:
    - *Content* = particular mutex.
    - A single mutex cannot be managed (*locked/unlocked*) by multiple `std::mutex` owners.

Move semantics

- Custom-defined operation for moving contents — move constructors and move assignment operators — typically make sense only for classes that represent ownership of some resources.
- *Example* — *unique pointer* owns a dynamically-allocated object (*resource*).
- Moving content then means that:
  - *destination object* takes over (acquires) the resource ownership,
  - *source object* gives up (releases) ownership of that resource.
- *Counter-example:*

```
struct Point2D {
  double x, y;
};
```

- Operation of moving content — *2D space point coordinates* — to another object logically does not make any sense.
- *Note* — `Point2D` is a trivially-copyable type.

Move semantics — reusability

- May "*moved-from*" objects be used?
  - = objects whose content has been moved from them.
  - The answer depends on how content-moving operations are implemented.
  - ⇒ *Advice* — always check with the documentation for a given (class) type (or, its source code).
- What about *C++ standard library types* that own resources (*other objects, strings, file streams, threads,…*)?
- *General rule of thumb:*
  - Moved-from objects may be used, but they are in valid but unspecified states (basically they have unspecified content).
  - ⇒ To use them, first, a new content need to be set/assigned to them.
  - C++ standard:
    - *Quote* — "*Unless otherwise specified, moved-from objects shall be placed in a valid but unspecified state.*"
    - *Current draft link* — [lib.types.movedfrom/1/3].

Move semantics — reusability *(cont.)*

- *Exception example* — `std::unique_ptr`:
  - *Moved-from unique pointer* is explicitly required to be in an "*empty state*" (ownership of no object).
  - *Current draft standard link* — [unique.ptr.single.ctor/20].
- Why does this not hold for all resource-owning library types?
  - Consider `std::string` implemented with SSO.
  - When owned string is *long*, moving content = "stealing" its pointer to that *heap-allocated string*.
    - ⇒ Here, it makes sense to put moved-from `std::string` object into an *empty state*.
  - But, when the owned string is *short*, it is stored in the *included storage* of the source (*moved-from*) owner.
  - ⇒ Here, the owned string characters needs to be transferred = copied into the included storage of the destination (*moved-to*) object.
  - Forcing *moved-from* object to be in *empty state* would actually make *content-moving* slower than *content-copying*.
    - Yet, implementations typically do that for sake of *backwards-portability*,… .
    - *Relevant discussion:* https://stackoverflow.com/q/52696413/580083.

Classes without content

- Some classes even do not represent any content.
- They may have various different purposes.
- *Example* — ***template metafunctions***:
  - Template metafunctions are class (struct) templates that “maps” template arguments to compile-time entities.
  - More particularly, they map *types or integer constants* to *types or integer constants*.
  - *Illustrative example* — metafunction definition (*above*), usage (*below*):

```
template <int I>                                    // metafunction parameter
struct increment_meta_fn {
  static /* constexpr */ const int value = I + 1;  // metafunction "result"
};
```

```
std::cout << increment_meta_fn<10>::value;  // prints out "11"
```

- Class template `increment_meta_fn` does not represent any content.
- It is even not intended to be *object-instantiated*:
  - There is no point in making objects of type `increment_meta_fn<I>`.

Type traits — introduction

- Template metafunctions make base of so-called ***type-traits library:***
  - It is a part of the C++ standard library.
  - It defines multiple metafunctions mostly for "treating" types in some sense.
- *Exemplary type-trait metafunction* — `std::is_trivially_copyable<T>`:
  - It maps a type (its template argument) to a *Boolean value* in the form of a *static constant* named `value`.
  - This constant is *true* or *false* if the input type *is* or *is not* trivially-copyable, respectively:

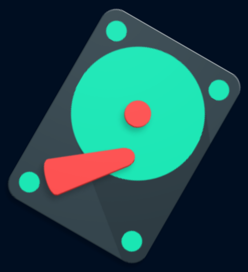
```cpp
#include <iostream>
#include <memory>
#include <type_traits>

int main() {
  std::cout << std::is_trivially_copyable< int*                 >::value;  // prints "1"
  std::cout << std::is_trivially_copyable< std::unique_ptr<int> >::value;  // prints "0"
}
```

- Metafunctions like the ones in type-traits library are mostly based *on template specialization* (*more details — further lectures*).

Trivial copyability and *serialization*

- *Object serialization:*
  - “sending”/storing object content into some “byte stream”.
- *Typical use cases:*
  - transmitting object content over a network,
  - storing object content into a file.
- *Counter-operation — deserialization:*
  - Reconstruction of object content from a “byte stream”.
- *Trivially-copyable types* — easy case:
  - *Serialization* of content = serialization of binary representation (copying its bytes into the target-output stream).
  - *Deserialization* = reverse action — copying bytes from byte stream into an object binary representation.
- *Non-trivially-copyable types* — more complicated:
  - Special custom-defined (de)serialization operations are required.



Other content-related operations

- Some other typical *content-related* operations:
- *Comparison of content:*
  - Typical implementation — `operator==`.
  - *Trivially-copyable types* = comparison of binary representations.
  - *Non-trivially-copyable types* = custom definitions required.
    - *Example* — `std::string` — `operator==` needs to compare owned strings (characters).
- *Swapping of content:*
  - Typical implementation — *free* or *member function* called swap.
  - *Trivially-copyable types* = swapping binary representations (bytes).
  - *Non-trivially-copyable types* = custom definitions required.
    - *Example* — `std::string` — swap needs to swap owned strings (wherever they are actually stored with respect to SSO).

Other content-related operations *(cont.)*

- *Setting "empty" content* (if this semantically makes sense):
  - Implementation — default constructor.
  - *Trivially-copyable types* — there is nothing as *empty content.*
    - *What is empty integer or floating-point number?*
  - *Non-trivially-copyable types* — typically makes sense mostly for resource-owning classes:
    - *Example* — `std::string` — empty string (no characters).
- *Setting content by conversion from object of a different type:*
  - Implementation — converting constructors.
    - *Example* — implicit conversion from `int` to `double`.
    - *Example* — `str::string` — conversion from a "*C-string*" (`char*` pointer).
- *Conversion of content into object of another type:*
  - Implementation — cast/conversion operators.
    - *Example* — `std::unique_ptr` / `std::optional` — cast to `bool` which is *true* if there is an owned object (*non-empty state*).

Initialization/assignment with content copying/moving — trivially-copyable cases

- *Initialization* with content copying/moving:
  - Initializes a new object and copies/moves its contents from the *initialization-expression object* of the same type.
- *Trivially-copyable types* — only content copying exist:

```
int i = 1;  // initialization with copying content (number 1) from literal "1" of type int

int j = i;  // initialization with copying content (number 1) from variable i of type int
int k(i);               // ...the very same effect
int l{i};               // ...the very same effect
int* ptr = new int(i);  // ...the very same effect
```

- *Assignment* with content copying/moving:
  - Copies/moves its contents from the *right-hand-side expression object* of the same type into an already existing object.
- *Trivially-copyable types* — again, only content copying exist:

```
int i = 1;           // initialization with copying content

int j;               // initialization to unspecified content (no copying)
int* ptr = new int;  // ...the very same effect

j = i;               // assignment with copying content from variable i of type int
*ptr = i;            // assignment with copying content from variable i of type int
```

Initialization/assignment with content copying/moving — non-trivially-copyable cases

- For *non-trivially-copyable types*, the situation is much more complicated.
- In *initialization*, *copy* and *move constructors* are involved.
- In *assignment*, *copy* and *move assignment operators* are involved.
- These *special member functions* are called when object is *initialized/assigned* and the *initialization/right-hand-side expression* is of the same type.
- If both *copy* and *move* operations exist, how to “select” which one should be used?

```
struct X {
  X();          // default constructor
  X(const X&);  // copy constructor
  X(X&&);       // move constructor
  ...
};
```

```
X x;

X x_copy( ??? );  // initialization with copying content from x
X x_move( ??? );  // initialization with moving content form x
```

Initialization/assignment with content copying/moving — non-trivially-copyable cases (*cont.*)

- *Simple solution example:*

```
X x;

X x_copy( x );
X x_move( std::move(x) );
```

- It looks simply but it is not!
- What does the utility library function `std::move` generally do?
  - First, it does not move object — such operation does not exist in C++.
  - Second, it does not even move content of object by itself.
  - In fact, it **does not move anything at all**.
  - Instead, it just changes the value category of its argument to *rvalue*.

- What does `std::move` effectively do in our example case?
  - By changing value category of *initialization expression* to *rvalue*, it causes move constructor of `std::string` to be used for initialization of s_move.
  - Why? To understand, we need to learn about *value categories* and *types of refences* (*see further lectures*).

Vector—realloc.—“*moving*” *elements (cont.)*

- Let’s get back to the “reallocation” of vector elements:

```
void reserve(size_t capacity) {
  if (capacity <= capacity_) return;
  T* data = (T*)::operator new(capacity * sizeof(T));  // new storage allocation

  for (size_t i = 0; i < size_; i++)
    new (data + i) T( ??? );  // initialization of size_ elements in new storage
```

- First, it should now be clear that we cannot use `memcpy` in `Vector` class — as we did in `String` class.
  - This would restrict `Vector` *value types* to trivially-copyable types only.
  - With `String`, use of `memcpy` was fine — `char` is a trivially-copyable type.
- What do we need?
  - *Original elements* have some content.
  - Reallocation is “*content-transparent*” ⇒ content of *original elements* needs to be preserved.
  - ⇒ *New initialized/constructed elements* need to have the same content as the *original* ones.

Vector—realloc.—"*moving*" *elements (cont.)*

- *"New initialized/constructed elements need to have the same content as the original ones."*
- $\Rightarrow$ For each *original element*, we need to *initialize new element* in the allocated storage with the same content.
- There are two content-related operations that can do that:
  - initialization with *content copying* semantics,
  - initialization with *content moving* semantics.
- After constructing *new elements* (in newly allocated storage), what will happen with the *original elements*?
  - They will be *destructed* (and their storage *released*).
- $\Rightarrow$ It makes sense to prefer content *moving* to *copying*.
  - Why to copy content if the original will be then destructed/released immediately?

Vector—realloc.—“*moving*” *elements (cont.)*

- *Conclusion* — we want to initialize new elements by preferring content moving.
- How to do this, when:
  - we don’t know whether the *vector value type* T is or is not *trivially-copyable*;
  - in case of it is not, we don’t know whether it has or has not available move and copy constructors.
- For *trivially-copyable types*, the solution would be simple (there is no content-moving):

```
for (size_t i = 0; i < size_; i++)
  new (data + i) T( *(data_ + i) );
```

  - ⇒ Initialization expression for each *new element* is the *original element*.
- But, for *non-trivially-copyable types*, this solution would cause problems, since it would always try to call **copy constructor**.
  - ⇒ *First*, it would effectively disallow objects of non-copyable types to be put into vectors.
  - ⇒ *Second*, it would be *inefficient*.

Vector—realloc.—“*moving*” *elements (cont.)*

```
for (size_t i = 0; i < size_; i++)
  new (data + i) T( *(data_ + i) );
```

- *Source of inefficiency* — after reallocation, the original elements need to be destructed (and their storage deallocated):

```
void clear() {
  for ( ; size_ > 0; size_--)
    (data_ + size_ - 1)->~T();
}

void reserve(size_t capacity) {
  if (capacity <= capacity_) return;
  T* data = (T*)::operator new(capacity * sizeof(T));

  for (size_t i = 0; i < size_; i++)
    new (data + i) T( *(data_ + i) );

  clear();                        // ending lifetime of original elements
  ::operator delete(data_);       // deallocation of original storage
  data_ = data; capacity_ = capacity;
}
```

- ⇒ With *this solution*, the content of the original elements is copied (which may be costly) and then immediately destructed/released.

Vector—realloc.—"*moving*" *elements (cont.)*

- More reasonable would be to "*try-to-move*" content from the original elements:

```
for (size_t i = 0; i < size_; i++)
  new (data + i) T( std::move( *(data_ + i) ) );
```

- In case of *non-trivially-copyable* vector value type with available move constructor, new elements will be initialized with this move constructor.
  - ⇒ The content will be moved from original to new elements.
- *Questions:*
  1) Will this work for *trivially-copyable types* that have no move-content semantics?
  2) Will this work for *non-trivially-copyable types* that do not have move constructor available?
- The answer for both question is (*fortunately*) **YES** :).

`std::move` — *no-move* cases

- *Ad 1)* For *trivially-copyable types*, "wrapping" *content-copying* initialization expression with `std::move` has absolutely no effect:

```
int i = 1;

int j = i;             // (1)
int k = std::move(i);  // (2) exactly the same effect as (1)
```

- *Ad 2)* For *non-trivially-copyable types* without available move constructor, "wrapping" *content-copying* initialization expression with `std::move` has no effect as well:

```
struct X {
  X();
  X(const X&);      // copy constructor available
                    // move constructor not available
};
```

```
int x1;

int x2 = x1;             // initialization by copy constructor
int x3 = std::move(x1);  // initialization by copy constructor as well
```

- *Explanation* — *binding of references* and corresponding *overloading rules* (*see further lectures*).

Vector—realloc.—*"moving" elements (cont.)*

- *Final* (?) solution:

```
void reserve(size_t capacity) {
  if (capacity <= capacity_) return;
  T* data = (T*)::operator new(capacity * sizeof(T));

  for (size_t i = 0; i < size_; i++)
    new (data + i) T( std::move( *(data_ + i) ) );

  clear();
  ::operator delete(data_);
  data_ = data; capacity_ = capacity;
}
```

- *Summary:*

1) *Trivial-copyable types:*
   - New elements are initialized by copying content (copying *binary representations*) of original elements.
2) *Non-trivial-copyable types:*
   - If move constructor is available, new elements are initialized by moving content from the original elements.
   - Otherwise, new elements are initialized by copying content of the original elements.
   - *Implication* — at least one of these constructors **needs to be available**!