CS101A(H) Week2

Class

Contents

Members of a class

A simple `class`

The initial idea: A `class` is a new kind of `struct` that can have member functions:

```cpp
class Student {
  std::string name;
  std::string id;
  int entranceYear;
  void setName(const std::string &newName) {
    name = newName;
  }
  void printInfo() const {
    std::cout << "I am " << name << ", id " << id
              << ", entrance year: " << entranceYear << std::endl;
  }
  bool graduated(int year) const {
    return year - entranceYear >= 4;
  }
};
```

Member access

Member access: `a.mem`, where `a` is an **object** of the class type.

- Every member [1] belongs to an object: each student has a name, id, entrance year, etc.
  - You need to specify *whose* name / id / ... you want to obtain.

To call a member function on an object: `a.memfun(args)`.

```
Student s = someValue();
s.printInfo(); // call its printInfo() to print related info
if (s.graduated(2023)) {
  // ...
}
```

Access control

```cpp
class Student {
private:
  std::string name;
  std::string id;
  int entranceYear;
public:
  void setName(const std::string &newName) { name = newName; }
  void printInfo() const {
    std::cout << "I am " << name << ", id " << id
              << ", entrance year: " << entranceYear << std::endl;
  }
  bool graduated(int year) const { return year - entranceYear >= 4; }
};
```

- `private` members: Only accessible to code inside the class and `friend` s.
  - $\Rightarrow$ We will introduce `friend` later.
- `public` members: Accessible to all parts of the program.

Access control

```cpp
class Student {
private:
  std::string name;
  std::string id;
  int entranceYear;

public:
  void setName(const std::string &newName);
  void printInfo() const;
  bool graduated(int year) const;
};
Student a;
a.entranceYear = 2025; // Error! It's private
a.printInfo(); // OK, It's public
```

Unlike some other languages (e.g. Java), an access specifier controls the access of all members after it, until the next access specifier or the end of the class definition.

Access control

```cpp
class Student {
// private:
  std::string name;
  std::string id;
  int entranceYear;
public:
  void setName(const std::string &newName);
  void printInfo() const;
  bool graduated(int year) const;
};
```

What if there is a group of members with no access specifier at the beginning?

- If it's `class`, they are `private`.
- If it's `struct`, they are `public`.

This is one of the **only two differences** between `struct` and `class` in C++.

Understand the `this` pointer

```cpp
class Student {
  // ...
public:
  bool graduated(int year) const;
};

Student s = someValue();
if (s.graduated(2023)) // ...
```

How many parameters does `graduated` have?

- **Seemingly one, but actually two:** `s` is also information that must be known when calling this function!

Understand the `this` pointer

```cpp
class Student {
public:
  void setName(const std::string &n) {
    name = n;
  }

  bool graduated(int year) const {
    return year - entranceYear >= 4;
  }
};

Student s = someValue();
if (s.graduated(2023))
  // ...
s.setName("Alice");
```

- The code on the left can be viewed as:

```cpp
void setName
    (Student *this, const std::string &n) {
  this->name = n;
}
bool graduated
    (const Student *this, int year) {
  return year - this->entranceYear >= 4;
}

Student s = someValue();
if (graduated(&s, 2023))
  // ...
setName(&s, "Alice");
```

Understand the `this` pointer

There is a pointer called `this` in each member function of class `X` which has type `X *` or `const X *`, pointing to the object on which the member function is called.

Inside a member function, access of any member `mem` is actually `this->mem`.

We can also write `this->mem` explicitly.

```
class Student {
public:
  bool graduated(int year) const {
    return year - this->entranceYear >= 4;
  }
};
```

Many languages have similar constructs, e.g. `self` in Python. (C++23 has `self` too!)

`const` member functions

The `const` keyword after the parameter list and before the function body `{` is used to declare a **`const` member function**.

- A `const` member function cannot modify its data members [2].
- A `const` member function **guarantees** that no data member will be modified.
  - A non-`const` member function does not provide such guarantee.
  - In a `const` member function, calling a non-`const` member function on `*this` is not allowed.
- For a `const` object, **only `const` member functions can be called on it**.

[Best practice] If, logically, a member function should not modify the object's state, it should be made a `const` member function. Otherwise, it cannot be called on `const` objects.

Examples that are not allowed in `const` member functions

```cpp
class Student {
public:
  void modifyMember() const {
    name += 'a'; // Error
  }
  void nonConstFunction() {}
  void constFunction() const {
    nonConstFunction(); // Error
  }
};
const Student a;
a.nonConstFunction(); // Error
```

`const` member functions and the `this` pointer

This `const` is essentially applied to the `this` pointer:

- In `const` member functions of class `X`, `this` has type `const X *`.
- In non-`const` member functions of class `X`, `this` has type `X *`.

If `ptr` is of type `const T *`, the expression `ptr->mem` is also `const`-qualified.

- Recall that in a member function, access of a member `mem` is actually `this->mem`.
- Therefore, `mem` is also `const`-qualified in a `const` member function.

```cpp
class Student {
public:
  void modifyMember() const {
    name += 'a'; // Error: `name` is `const std::string` in a const member
                 // function. It cannot be modified.
  }
};
```

`const` member functions

Effective C++ Item 3: **Use `const` whenever possible.**

Decide whether the following member functions need a `const` qualification:

```cpp
class Student {
  std::string name, id;
  int entranceYear;
public:
  const std::string &getName(); // returns the name of the student.
  const std::string &getID();   // returns the id of the student.
  bool valid();    // verifies whether the leading four digits in `id`
                   // is equal to `entranceYear`.
  void adjustID(); // adjust `id` according to `entranceYear`.
};
```

`const` member functions

Effective C++ Item 3: **Use `const` whenever possible.**

Decide whether the following member functions need a `const` qualification:

```cpp
class Student {
  std::string name, id;
  int entranceYear;
public:
  const std::string &getName() const; // returns the name of the student.
  const std::string &getID() const;   // returns the id of the student.
  bool valid() const;    // verifies whether the leading four digits in `id`
                         // is equal to `entranceYear`.
  void adjustID(); // adjust `id` according to `entranceYear`.
};
```

The `const`ness of member functions should be determined **logically**.

Constructors

Often abbreviated as "ctors".

Constructors

Constructors define how an object can be initialized. And Constructors are often **overloaded**, because an object may have multiple reasonable ways of initialization.

```cpp
class Student {
  std::string name;
  std::string id;
  int entranceYear;
public:
  Student(const std::string &name_, const std::string &id_, int ey)
    : name(name_), id(id_), entranceYear(ey) {}
  Student(const std::string &name_, const std::string &id_)
    : name(name_), id(id_), entranceYear(std::stoi(id_.substr(0, 4))) {}
};

Student a("Alice", "2020123123", 2020);
Student b("Bob", "2020123124"); // entranceYear = 2020
Student c; // Error: No default constructor. (to be discussed later)
```

Constructors

```
class Student {
  std::string name;
  std::string id;
  int entranceYear;

public:
  Student(const std::string &name_, const std::string &id_)
    : name(name_), id(id_), entranceYear(std::stoi(id_.substr(0, 4))) {}
};
```

- The constructor name is the class name: `Student`.
- Constructors do not have a return type (not even `void` [3]). The constructor body can contain a `return;` statement, which should not return a value.
- The function body of this constructor is empty: `{}`.

Constructor initializer list

Constructors initialize **all data members** of the object.

The initialization of **all data members** is done **before entering the function body**.

How they are initialized is (partly) determined by the **constructor initializer list**:

```cpp
class Student {
  // ...
public:
  Student(const std::string &name_, const std::string &id_)
    : name(name_), id(id_), entranceYear(std::stoi(id_.substr(0, 4))) {}
};
```

The initializer list starts with `:`, and contains initializers for each data member, separated by `,`. The initializers must be of the form `(...)` or `{...}`, not `= ...`.

Order of initialization

Data members are initialized in order **in which they are declared**, not the order in the initializer list.

- If the initializers appear in an order different from the declaration order, the compiler will generate a warning.

Typical mistake: `entranceYear` is initialized in terms of `id`, but `id` is not initialized yet!

```
class Student {
  std::string name;
  int entranceYear; // !!!
  std::string id;

public:
  Student(const std::string &name_, const std::string &id_)
    : name(name_), id(id_), entranceYear(std::stoi(id.substr(0, 4))) {}
};
```

Constructor initializer list

Data members are initialized in order **in which they are declared**, not the order in the initializer list.

- If the initializers appear in an order different from the declaration order, the compiler will generate a warning.
- For a data member that do not appear in the initializer list:
  - If there is an **in-class initializer** (see next page), it is initialized using the in-class initializer.
  - Otherwise, it is **default-initialized**.

What does **default-initialization** mean for class types? $\Rightarrow$ To be discussed later.

In-class initializers

A member can have an in-class initializer. It must be in the form `{...}` or `= ...`.[4]

```
class Student {
  std::string name = "Alice";
  std::string id;
  int entranceYear{2024}; // equivalent to `int entranceYear = 2024;`.
public:
  Student() {} // `name` is initialized to `"Alice"`,
               // `id` is initialized to an empty string,
               // and `entranceYear` is initialized to 2024.
  Student(int ey) : entranceYear(ey) {} // `name` is initialized to `"Alice"`,
                                        // `id` is initialized to an empty string,
                                        // and `entranceYear` is initialized to `ey`.
};
```

The in-class initializer provides the "default" way of initializing a member in this class, as a substitute for default-initialization.

Constructor initializer list

Below is a typical way of writing this constructor without an initializer list:

```cpp
class Student {
  // ...
public:
  Student(const std::string &name_, const std::string &id_) {
    name = name_;
    id = id_;
    entranceYear = std::stoi(id_.substr(0, 4));
  }
};
```

How are these members actually initialized in this constructor?

Constructor initializer list

Below is a typical way of writing this constructor without an initializer list:

```cpp
class Student {
  // ...
public:
  Student(const std::string &name_, const std::string &id_) {
    name = name_;
    id = id_;
    entranceYear = std::stoi(id_.substr(0, 4));
  }
};
```

How are these members actually initialized in this constructor?

- First, before entering the function body, `name`, `id` and `entranceYear` are default-initialized. `name` and `id` are initialized to empty strings.
- Then, the assignments in the function body take place.

Constructor initializer list

[Best practice] <u>Always use an initializer list in a constructor.</u>

- Not all types can be default-initialized. Not all types can be assigned to. (Any counterexamples?)

Constructor initializer list

[Best practice] Always use an initializer list in a constructor.

Not all types can be default-initialized. Not all types can be assigned to.

- References `T &` cannot be default-initialized, and cannot be assigned to.
- `const` objects of built-in types cannot be default-initialized.
- `const` objects cannot be assigned to.
- A class can choose to allow or disallow default initialization or assignment. It depends on the design. $\Rightarrow$ See next page.

Moreover, if a data member is default-initialized and then assigned when could have been initialized directly, it may lead to low efficiency.

Default constructors

A special constructor that takes no parameters.

- Guess what it's for?

Default Constructors

A special constructor that takes no parameters.

- It defines the behavior of **default-initialization** of objects of that class type, since no arguments need to be passed when calling it.

```cpp
class Point2d {
  double x, y;
public:
  Point2d() : x(0), y(0) {} // default constructor
  Point2d(double x_, double y_) : x(x_), y(y_) {}
};

Point2d p1;       // calls default ctor, (0, 0)
Point2d p2(3, 4); // calls Point2d(double, double), (3, 4)
Point2d p3();     // Is this calling the default ctor?
```

Default constructors

A special constructor that takes no parameters.

- It defines the behavior of **default-initialization** of objects of that class type, since no arguments need to be passed when calling it.

```cpp
class Point2d {
  double x, y;
public:
  Point2d() : x(0), y(0) {} // default constructor
  Point2d(double x_, double y_) : x(x_), y(y_) {}
};

Point2d p1;       // calls default ctor, (0, 0)
Point2d p2(3, 4); // calls Point2d(double, double), (3, 4)
Point2d p3();     // Is this calling the default ctor?
```

Be careful! `p3` is a **function** that takes no parameters and returns `Point2d`.

Lifetime, constructor and destructor

Lifetime of an object

Lifetime of a local non-`static` object:

- Starts on initialization
- Ends when control flow goes out of its **scope**.

```
for (int i = 0; i != n; ++i) {
  do_something(i);
  // Lifetime of `s` begins.
  std::string s = some_string();
  do_something_else(s, i);
/* end of lifetime of `s` */ }
```

Every time the loop body is executed, `s` undergoes initialization and destruction.

- `std::string` **owns** some resources (memory where the characters are stored).
- `std::string` must *somehow* release that resources (deallocate that memory) at the end of its lifetime.

Lifetime of an object

Lifetime of a global object:

- Starts on initialization (before the first statement of `main`)
- Ends when the program terminates.

Lifetime of a heap-based object:

- Starts on initialization: **A `new` expression will do this, but `malloc` does not!**
- Ends when it is destroyed: **A `delete` expression will do this, but `free` does not!**

Constructors and Destructors

Take `std::string` as an example:

- Its initialization (done by its constructors) must allocate some memory for its content.
- When it is destroyed, it must *somehow* deallocate that memory.

Constructors and Destructors

Take `std::string` as an example:

- Its initialization (done by its constructors) must allocate some memory for its content.
- When it is destroyed, it must *somehow* deallocate that memory.

A destructor of a class is the function that is automatically called when an object of that class type is destroyed.

Constructors and Destructors

Syntax of destructors: `~ClassName() { /* ... */ }`

```cpp
struct A {
  A() {
    std::cout << 'c';
  }
  ~A() {
    std::cout << 'd';
  }
};
```

```cpp
for (int i = 0; i != 3; ++i) {
  A a;
  // do something ...
}
```

Output:

```
cdcdcd
```

Destructor

Called **automatically** when the object is destroyed!

- How can we make use of this property?

Destructor

Called **automatically** when the object is destroyed!

- How can we make use of this property?

We often do some **cleanup** in a destructor:

- If the object **owns some resources** (e.g. dynamic memory), destructors can be made use of to avoid leaking!

```cpp
class A {
  SomeResourceHandle resource;

public:
  A(/* ... */) : resource(obtain_resource(/* ... */)) {}
  ~A() {
    release_resource(resource);
  }
};
```

Example: A dynamic array

Suppose we want to implement a "dynamic array":

- It looks like a VLA (variable-length array), but it is heap-based, which is safer.
- It should take good care of the memory it uses.

Expected usage:

```
int n; std::cin >> n;
Dynarray arr(n); // `n` is runtime determined
                 // `arr` should have allocated memory for `n` `int`s now.
for (int i = 0; i != n; ++i) {
  int x; std::cin >> x;
  arr.at(i) = x * x; // subscript, looks as if `arr[i] = x * x`
}
// ...
// `arr` should deallocate its memory itself.
```

Dynarray: members

- It should have a pointer that points to the memory, where elements are stored.
- It should remember its length.

```
class Dynarray {
  int *m_storage;
  std::size_t m_length;
};
```

- `m` stands for **member**.

[Best practice] Make data members `private`, to achieve good encapsulation.

Dynarray: constructors

- We want `Dynarray a(n);` to construct a `Dynarray` that contains `n` elements.
  - To avoid troubles, we want the elements to be **value-initialized**!
    - **Value-initialization** is like "empty-initialization" in C.
  - `new int[n]{}`: Allocate a block of heap memory that stores `n` `int`s, and value-initialize them.
- Do we need a default constructor?
  - Review: What is a default constructor?
    - The constructor with no parameters.
  - What should be the correct behavior of it?

Dynarray: constructors

- We want `Dynarray a(n);` to construct a `Dynarray` that contains `n` elements.
  - To avoid troubles, we want the elements to be **value-initialized**!
- Suppose we don't want a default constructor.

```cpp
class Dynarray {
  int *m_storage;
  std::size_t m_length;
 public:
  Dynarray(std::size_t n) : m_storage(new int[n]{}), m_length(n) {}
};
```

If the class has a user-declared constructor, the compiler will not generate a default constructor.

Dynarray: constructors

```
class Dynarray {
  int *m_storage;
  std::size_t m_length;
 public:
  Dynarray(std::size_t n) : m_storage(new int[n]{}), m_length(n) {}
};
```

Since `Dynarray` has a user-declared constructor, it does not have a default constructor:

```
Dynarray a; // Error.
```

Dynarray: destructor

- Remember: The destructor is (automatically) called when the object is "dead".
- The memory is obtained in the constructor, and released in the destructor.

```cpp
class Dynarray {
  int *m_storage;
  std::size_t m_length;
 public:
  Dynarray(std::size_t n)
    : m_storage(new int[n]{}), m_length(n) {}
  ~Dynarray() {
    delete[] m_storage; // Pay attention to `[]`!
  }
};
```

Dynarray: some member functions

Design some useful member functions.

- A function to obtain its length (size).
- A function telling whether it is empty.

```
class Dynarray {
  // ...
 public:
  std::size_t size() const {
    return m_length;
  }
  bool empty() const {
    return m_length == 0;
  }
};
```

Dynarray: some member functions

Design some useful member functions.

- A function returning **reference** to an element.

```
class Dynarray {
  // ...
 public:
  int &at(std::size_t i) {
    return m_storage[i];
  }
  const int &at(std::size_t i) const {
    return m_storage[i];
  }
};
```

Why do we need this "`const` vs non-`const`" overloading?

Dynarray: Usage

```cpp
void print(const Dynarray &a) {
  for (std::size_t i = 0;
       i != a.size(); ++i)
    std::cout << a.at(i) << ' ';
  std::cout << std::endl;
}
void reverse(Dynarray &a) {
  for (std::size_t i = 0,
    j = a.size() - 1; i < j; ++i, --j)
    std::swap(a.at(i), a.at(j));
}
```

```cpp
int main() {
  int n; std::cin >> n;
  Dynarray array(n);
  for (int i = 0; i != n; ++i)
    std::cin >> array.at(i);
  reverse(array);
  print(array);
  return 0;
  // Dtor of `array` is called here,
  // which deallocates the memory
}
```

Copy control

Copy-initialization

We can easily construct a `std::string` to be a copy of another:

```
std::string s1 = some_value();
std::string s2 = s1; // s2 is initialized to be a copy of s1
std::string s3(s1); // equivalent
std::string s4{s1}; // equivalent, but modern
```

Can we do this for our `Dynarray` ?

Copy-initialization

Before we add anything, let's try what will happen:

```
Dynarray a(3);
a.at(0) = 2; a.at(1) = 3; a.at(2) = 5;
Dynarray b = a; // It compiles.
print(b); // 2 3 5
a.at(0) = 70;
print(b); // 70 3 5
```

Ooops! Although it compiles, the pointers `a.m_storage` and `b.m_storage` are pointing to the same address!

Copy-initialization

Before we add anything, let's try what will happen:

```
Dynarray a(3);
Dynarray b = a;
```

Although it compiles, the pointers `a.m_storage` and `b.m_storage` are pointing to the same address!

This will cause disaster: consider the case if `b` "dies" before `a`:

```
Dynarray a(3);
if (some_condition) {
  Dynarray b = a; // `a.m_storage` and `b.m_storage` point to the same memory!
  // ...
} // At this point, dtor of `b` is invoked, which deallocates the memory.
std::cout << a.at(0); // Invalid memory access!
```

Copy constructor

Let `a` be an object of type `Type`. The behaviors of **copy-initialization** (in one of the following forms)

```
Type b = a;
Type b(a);
Type b{a};
```

are determined by a constructor: **the copy constructor**.

- Note! The `=` in `Type b = a;` **is not an assignment operator**!

Copy constructor

The copy constructor of a class `X` has a parameter of type `const X &`:

```
class Dynarray {
 public:
  Dynarray(const Dynarray &other);
};
```

Why `const`?

- Logically, it should not modify the object being copied.

Why `&`?

- **Avoid copying.** Pass-by-value is actually **copy-initialization** of the parameter, which will cause infinite recursion here!

Dynarray: copy constructor

What should be the correct behavior of it?

```cpp
class Dynarray {
 public:
  Dynarray(const Dynarray &other);
};
```

Dynarray: copy constructor

- We want a copy of the content of `other`.

```
class Dynarray {
 public:
  Dynarray(const Dynarray &other)
    : m_storage(new int[other.size()]{}), m_length(other.size()) {
    for (std::size_t i = 0; i != other.size(); ++i)
      m_storage[i] = other.at(i);
  }
};
```

Now the copy-initialization of `Dynarray` does the correct thing:

- The new object allocates a new block of memory.
- The **contents** are copied, not just the address.

Synthesized copy constructor

If the class does not have a user-declared copy constructor, the compiler will try to synthesize one:

- The synthesized copy constructor will **copy-initialize** all the members, as if

  ```
  class Dynarray {
   public:
    Dynarray(const Dynarray &other)
      : m_storage(other.m_storage), m_length(other.m_length) {}
  };
  ```

- If the synthesized copy constructor does not behave as you expect, **define it on your own!**

Defaulted copy constructor

If the synthesized copy constructor behaves as we expect, we can explicitly require it:

```
class Dynarray {
 public:
  Dynarray(const Dynarray &) = default;
  // Explicitly defaulted: Explicitly requires the compiler to synthesize
  // a copy constructor, with default behavior.
};
```

Deleted copy constructor

What if we don't want a copy constructor?

```cpp
class ComplicatedDevice {
  // some members
  // Suppose this class represents some complicated device,
  // for which there is no correct and suitable behavior for "copying".
};
```

Simply not defining the copy constructor does not work:

- The compiler will synthesize one for you.

Deleted copy constructor

What if we don't want a copy constructor?

```
class ComplicatedDevice {
  // some members
  // Suppose this class represents some complicated device,
  // for which there is no correct and suitable behavior for "copying".
 public:
  ComplicatedDevice(const ComplicatedDevice &) = delete;
};
```

By saying `= delete`, we define a **deleted** copy constructor:

```
ComplicatedDevice a = something();
ComplicatedDevice b = a; // Error: calling deleted function
```

Copy-assignment operator

Apart from copy-initialization, there is another form of copying:

```cpp
std::string s1 = "hello", s2 = "world";
s1 = s2; // s1 becomes a copy of s2, representing "world"
```

In `s1 = s2`, `=` is the **assignment operator**.

`=` is the assignment operator **only when it is in an expression.**

- `s1 = s2` is an expression.
- `std::string s1 = s2` is in a **declaration statement**, not an expression. `=` here is a part of the initialization syntax.

Dynarray: copy-assignment operator

The copy-assignent operator is defined in the form of **operator overloading**:

- `a = b` is equivalent to `a.operator=(b)` .
- We will not talk about operator overloading in lectures.

```
class Dynarray {
 public:
  Dynarray &operator=(const Dynarray &other);
};
```

- The function name is `operator=` .
- In consistent with built-in assignment operators, `operator=` returns **reference to the left-hand side object** (the object being assigned).
  - It is `*this` .

Dynarray: copy-assignment operator

We also want the copy-assignment operator to copy the contents, not only an address.

```
class Dynarray {
 public:
  Dynarray &operator=(const Dynarray &other) {
    m_storage = new int[other.size()];
    for (std::size_t i = 0; i != other.size(); ++i)
      m_storage[i] = other.at(i);
    m_length = other.size();
    return *this;
  }
};
```

Is this correct?

Dynarray: copy-assignment operator

Avoid memory leaks! Deallocate the memory you don't use!

```
class Dynarray {
 public:
  Dynarray &operator=(const Dynarray &other) {
    delete[] m_storage; // !!!
    m_storage = new int[other.size()];
    for (std::size_t i = 0; i != other.size(); ++i)
      m_storage[i] = other.at(i);
    m_length = other.size();
    return *this;
  }
};
```

Is this correct?

Dynarray: copy-assignment operator

What if **self-assignment** happens?

```cpp
class Dynarray {
 public:
  Dynarray &operator=(const Dynarray &other) {
    // If `other` and `*this` are actually the same object,
    // the memory is deallocated and the data are lost! (DISASTER)
    delete[] m_storage;
    m_storage = new int[other.size()];
    for (std::size_t i = 0; i != other.size(); ++i)
      m_storage[i] = other.at(i);
    m_length = other.size();
    return *this;
  }
};
```

Dynarray: copy-assignment operator

Assignment operators should be **self-assignment-safe**.

```
class Dynarray {
 public:
  Dynarray &operator=(const Dynarray &other) {
    int *new_data = new int[other.size()];
    for (std::size_t i = 0; i != other.size(); ++i)
      new_data[i] = other.at(i);
    delete[] m_storage;
    m_storage = new_data;
    m_length = other.size();
    return *this;
  }
};
```

This is self-assignment-safe. (Think about it.)

Synthesized, defaulted and deleted copy-assignment operator

Like the copy constructor:

- The copy-assignment operator can also be **deleted**, by declaring it as `= delete;`.
- If you don't define it, the compiler will generate one that copy-assigns all the members, as if it is defined as:

```
class Dynarray {
 public:
  Dynarray &operator=(const Dynarray &other) {
    m_storage = other.m_storage;
    m_length = other.m_length;
    return *this;
  }
};
```

- You can also require a synthesized one explicitly by saying `= default;`.

[IMPORTANT] The rule of three: Reasoning

Among the **copy constructor**, the **copy-assignment operator** and the **destructor**:

- If a class needs a user-provided version of one of them, **usually**, it needs a user-provided version of **each** of them.
- Why?

[IMPORTANT] The rule of three: Reasoning

Among the **copy constructor**, the **copy-assignment operator** and the **destructor**:

- If a class needs a user-provided version of one of them,
- **usually**, it is a class that **manages some resources**,
- for which **the default behavior of the copy-control members does not suffice**.
- Therefore, all of the three special functions need a user-provided version.
  - Define them in a correct, well-defined manner.
  - If a class should not be copy-constructible or copy-assignable, **delete that function**.

Type alias members

Type aliases in C++: `using`.

A better way of declaring type aliases:

```
// C-style
typedef long long LL;
// C++-style
using LL = long long;
```

It is more readable when dealing with compound types:

```
// C-style
typedef int intarray_t[1000];
// C++-style
using intarray_t = int[1000];
```

```
// C-style
typedef int (&ref_to_array)[1000];
// C++-style
using ref_to_array = int (&)[1000];
```

[Best practice] In C++, Use `using` to declare type aliases.

Type alias members

A class can have **type alias members**.

```cpp
class Dynarray {
 public:
  using size_type = std::size_t;
  size_type size() const { return m_length; }
};
```

Usage: `ClassName::TypeAliasName`

```cpp
for (Dynarray::size_type i = 0; i != a.size(); ++i)
  // ...
```

Note: Here we use `ClassName::` instead of `object.`, because such members belong to **the class**, not one single object.

Type alias members

The class also has control over the accessibility of type alias members.

```
class A {
  using type = int;
};
A::type x = 42; // Error: Accessing private member of `A`.
```

The class has control over the accessibility of **anything that is called a *member* of it**.

Type alias members in the standard library

All standard library containers (and `std::string`) define the type alias member `size_type` as the return type of `.size()`:

```
std::string::size_type i = s.size();
std::vector<int>::size_type j = v.size(); // Not `std::vector::size_type`!
                                          // The template argument `<int>`
                                          // is necessary here.
std::list<int>::size_type k = l.size();
```

Why?

Type alias members in the standard library

All standard library containers (and `std::string`) define the type alias member `size_type` as the return type of `.size()`:

```
std::string::size_type i = s.size();
std::vector<int>::size_type j = v.size();
std::list<int>::size_type k = l.size();
```

- This type is **container-dependent**: Different containers may choose different types suitable for representing sizes.
  - The Qt containers often use `int` as `size_type`.
- Define `Container::size_type` to achieve good **consistency** and **generality**.

`static` members

`static` data members

A `static` data member:

```
class A {
  static int something;
  // other members ...
};
```

Just consider it as a **global variable**, except that

- its name is in the **class scope**: `A::something`, and that
- the accessibility may be restricted. Here `something` is `private`.

`static` data members

A `static` data member:

```
class A {
  static int something;
  // other members ...
};
```

There is **only one** `A::something`: it does not belong to any object of `A`. It belongs to the **class** `A`.

- Like type alias members, we use `ClassName::` instead of `object.` to access them.

`static` data members

A `static` data member:

```cpp
class A {
  static int something;
  // other members ...
};
```

It can also be accessed by `a.something` (where `a` is an object of type `A`), but `a.something` and `b.something` refer to the same variable.

- If `f` is a function that returns an object of type `A`, `f().something` always accesses the same variable no matter what `f()` returns.
- In the very first externally available C++ compiler (Cfront 1.0, 1985), `f` in the expression `f().something` is not even called! This bug has been fixed soon.

`static` data members: Example

Suppose we want to assign a unique id to each object of our class.

```cpp
int cnt = 0;

class Dynarray {
  int *m_storage;
  std::size_t m_length;
  int m_id;
public:
  Dynarray(std::size_t n)
      : m_storage(new int[n]{}), m_length(n), m_id(cnt++) {}
  Dynarray() : m_storage(nullptr), m_length(0), m_id(cnt++) {}
  // ...
};
```

We use a global variable `cnt` as the "counter". Is this a good design?

`static` data members: Example

The name `cnt` is confusing: A "counter" of what?

```cpp
int X_cnt = 0, Y_cnt = 0, Z_cnt = 0;
struct X {
  int m_id;
  X() : m_id(X_cnt++) {}
};
struct Y {
  int m_id;
  Y() : m_id(Y_cnt++) {}
};
struct Z {
  int m_id;
  Z() : m_id(Z_cnt++) {}
};
```

- The program is in a mess with global variables all around.
- No prevention from potential mistakes:

  ```cpp
  struct Y {
    Y() : m_id(X_cnt++) {}
  };
  ```

  The mistake happens silently.

`static` data members: Example

Restrict the name of this counter in the scope of the corresponding class, by declaring it as a `static` data member.

- This is exactly the idea behind `static` data members: A "global variable" restricted in class scope.

```cpp
class Dynarray {
  static int s_cnt; // !!!
  int *m_storage;
  std::size_t m_length;
  int m_id;

public:
  Dynarray(/* ... */) : /* ... */, m_id(s_cnt++) {}
};
```

- `s` stands for `static`.

`static` data members

```cpp
class Dynarray {
  static int s_cnt; // !!!
  int *m_storage;
  std::size_t m_length;
  int m_id;

public:
  Dynarray(/* ... */) : /* ... */, m_id(s_cnt++) {}
};
```

You also need to give it a definition outside the class, according to some rules.

```cpp
int Dynarray::s_cnt; // Zero-initialize, because it is `static`.
```

Or initialize it with some value explicitly:

```cpp
int Dynarray::s_cnt = 42;
```

`static` data members

Exercise: `std::string` has a `find` member function:

```cpp
std::string s = something();
auto pos = s.find('a');
if (pos == std::string::npos) { // This means that `'a'` is not found.
  // ...
} else {
  std::cout << s[pos] << '\n'; // If executed, it should print `a`.
}
```

`std::string::npos` is returned when the required character is not found.

Define `npos` and `find` for your `Dynarray` class, whose behavior should be similar to those of `std::string`.

`static` member functions

A `static` member function:

```
class A {
 public:
  static void fun(int x, int y);
};
```

Just consider it as a normal non-member function, except that

- its name is in the **class scope**: `A::fun(x, y)`, and that
- the accessibility may be restricted. Here `fun` is `public`.

`static` member functions

A `static` member function:

```
class A {
 public:
  static void fun(int x, int y);
};
```

`A::fun` does not belong to any object of `A`. It belongs to the **class** `A`.

- There is no `this` pointer inside `fun`.

It can also be called by `a.fun(x, y)` (where `a` is an object of type `A`), but here `a` will not be bound to a `this` pointer, and `fun` has no way of accessing any non-`static` member of `a`.

`friend`

`friend` functions

Recall the `Student` class:

```cpp
class Student {
  std::string m_name;
  std::string m_id;
  int m_entranceYear;
public:
  Student(const std::string &name, const std::string &id)
      : m_name(name), m_id(id), m_entranceYear(std::stol(id.substr(0, 4))) {}
  auto graduated(int year) const { return year - m_entranceYear >= 4; }
  // ...
};
```

Suppose we want to write a function to display the information of a `Student`.

`friend` functions

```cpp
void print(const Student &stu) {
  std::cout << "Name: " << stu.m_name << ", id: " << stu.m_id
            << "entrance year: " << stu.m_entranceYear << '\n';
}
```

This won't compile, because `m_name`, `m_id` and `m_entranceYear` are `private` members of `Student`.

- One workaround is to define `print` as a member of `Student`.
- However, there do exist some functions that cannot be defined as a member.

`friend` functions

Add a `friend` declaration, so that `print` can access the private members of `Student`.

```
class Student {
  friend void print(const Student &); // The parameter name is not used in this
                                      // declaration, so it is omitted.

  std::string m_name;
  std::string m_id;
  int m_entranceYear;
public:
  Student(const std::string &name, const std::string &id)
      : m_name(name), m_id(id), m_entranceYear(std::stol(id.substr(0, 4))) {}
  auto graduated(int year) const { return year - m_entranceYear >= 4; }
  // ...
};
```

What if we use or not use `friend` functions

```cpp
class Student {
  friend void print(const Student &); // friend function declaration

  // ...
};
void print(const Student &a) { // friend function definition
  std::cout << a.m_name; // OK, friend function can access private members
}
void printNotFriend(const Student &a) {
  std::cout << a.m_name; // Error!
}
```

`friend` functions

```cpp
class Student {
  friend void print(const Student &);

  // ...
};
```

A `friend` is **not** a member! You can put this `friend` delcaration **anywhere in the class body**. The access modifiers have **no effect** on it.

- We often declare all the `friend`s of a class in the beginning or at the end of class definition.

`friend` classes

A class can also declare another class as its `friend`.

```
class X {
  friend class Y;
  // ...
};
```

In this way, any code from the class `Y` can access the private members of `X`.

Definition and declaration

Definition and declaration

For a function:

```cpp
// Only a declaration: The function body is not present.
void foo(int, const std::string &);
// A definition: The function body is present.
void foo(int x, const std::string &s) {
  // ...
}
```

Class definition

For a class, a **definition** consists of **the declarations of all its members**.

```cpp
class Widget {
public:
  Widget();
  Widget(int, int);
  void set_handle(int);

  // `const` is also a part of the function type, which should be present
  // in its declaration.
  const std::vector<int> &get_gadgets() const;

  // ...
private:
  int m_handle;
  int m_length;
  std::vector<int> m_gadgets;
};
```

Define a member function outside the class body

A member function can be declared in the class body, and then defined outside.

```cpp
class Widget {
public:
  const std::vector<int> &get_gadgets() const; // A declaration only.
  // ...
}; // Now the definition of `Widget` is complete.

// Define the function here. The function name is `Widget::get_gadgets`.
const std::vector<int> &Widget::get_gadgets() const {
  return m_gadgets; // Just like how you do it inside the class body.
                    // The implicit `this` pointer is still there.
}
```

The `::` operator

```
class Widget {
public:
  using gadgets_list = std::vector<int>;
  static int special_member;
  const gadgets_list &get_gadgets() const;
  // ...
};
const Widget::gadgets_list &Widget::get_gadgets() const {
  return m_gadgets;
}
```

- The members `Widget::gadgets_list` and `Widget::special_member` are accessed through `ClassName::`.
- The name of the member function `get_gadgets` is `Widget::get_gadgets`.

Class declaration and incomplete type

To declare a class without providing a definition:

```
class A;
struct B;
```

If we only see the **declaration** of a class, we have no knowledge about its members, how many bytes it takes, how it can be initialized, ...

- Such class type is an **incomplete type**.
- We cannot create an object of such type, nor can we access any of its members.
- The only thing we can do is to declare a pointer or a reference to it.

Class declaration and incomplete type

If we only see the **declaration** of a class, we have no knowledge about its members, how many bytes it takes, how it can be initialized, ...

- Such class type is an **incomplete type**.
- We cannot create an object of such type, nor can we access any of its members.
- The only thing we can do is to declare a pointer or a reference to it.

```
class Student; // We only have this declaration.

void print(const Student &stu) { // OK. Declaring a reference to it is OK.
  std::cout << stu.getName(); // Error. We don't know anything about its members.
}

class Student {
public:
  const std::string &getName() const { /* ... */ }
  // ...
```

Destructors revisited

Destructors revisited

A **destructor** (dtor) is a member function that is called automatically when an object of that class type is "dead".

- For global and `static` objects, on termination of the program.
- For local objects, when control reaches the end of its scope.
- For objects created by `new` / `new[]`, when their address is passed to `delete` / `delete[]`.

The destructor is often responsible for doing some **cleanup**: Release the resources it owns, do some logging, cut off its connection with some external objects, ...

Destructors

```
class Student {
  std::string m_name;
  std::string m_id;
  int m_entranceYear;
public:
  Student(const std::string &, const std::string &);
  const std::string &getName() const;
  bool graduated(int) const;
  void setName(const std::string &);
  void print() const;
};
```

Does our `Student` class have a destructor?

Destructors

Does our `Student` class have a destructor?

- It **must** have. Whenever you create an object of type `Student`, its destructor needs to be invoked somewhere in this program. [1]

What does `Student::~Student` need to do? Does `Student` own any resources?

Destructors

Does our `Student` class have a destructor?

- It **must** have. Whenever you create an object of type `Student`, its destructor needs to be invoked somewhere in this program. [5]

What does `Student::~Student` need to do? Does `Student` own any resources?

- It seems that a `Student` has no resources, so nothing special needs to be done.
- However, it has two `std::string` members! Their destructors must be called, otherwise the memory is leaked!

Destructors

To define the destructor of `Student`: Just write an empty function body, and everything is done.

```cpp
class Student {
  std::string m_name;
  std::string m_id;
  int m_entranceYear;
public:
  ~Student() {}
};
```

Destructors

```cpp
class Student {
  std::string m_name;
  std::string m_id;
  int m_entranceYear;
public:
  ~Student() {}
};
```

- When the function body is executed, the object is *not yet* "dead".
  - You can still access its members.

    ```cpp
    ~Student() { std::cout << m_name << '\n'; }
    ```

- After the function body is executed, **all its data members** are destroyed automatically, **in reverse order** in which they are declared.
  - For members of class type, their destructors are invoked automatically.

Constructors vs destructors

```
Student(const std::string &name)
    : m_name(name) /* ... */ {
  // ...
}
```

- A class may have multiple ctors (overloaded).
- The data members are initialized **before** the execution of function body.
- The data members are initialized **in order** in which they are declared.

```
~Student() {
  // ...
}
```

- A class has only one dtor. [6]
- The data members are destroyed **after** the execution of function body.
- The data members are destroyed **in reverse order** in which they are declared.

Compiler-generated destructors

For most cases, a class needs a destructor.

Therefore, the compiler always generates one [7] if there is no user-declared destructor.

- The compiler-generated destructor is `public` by default.
- The compiler-generated destructor is as if it were defined with an empty function body `{}`.
- It does nothing but to destroy the data members.

We can explicitly require one by writing `= default;`, just as for other copy control members.

Rvalue references

Motivation: Copy is slow.

```
std::string a = some_value(), b = some_other_value();
std::string s;
s = a;
s = a + b;
```

Consider the two assignments: `s = a` and `s = a + b`.

How is `s = a + b` evaluated?

Motivation: Copy is slow.

```
s = a + b;
```

1. Evaluate `a + b` and store the result in a temporary object, say `tmp`.
2. Perform the assignment `s = tmp`.
3. The temporary object `tmp` is no longer needed, hence destroyed by its destructor.

Can we make this faster?

Motivation: Copy is slow.

```
s = a + b;
```

1. Evaluate `a + b` and store the result in a temporary object, say `tmp`.
2. Perform the assignment `s = tmp`.
3. The temporary object `tmp` is no longer needed, hence destroyed by its destructor.

Can we make this faster?

- The assignment `s = tmp` is done by **copying** the contents of `tmp`?
- But `tmp` is about to "die"! Why can't we just *steal* the contents from it?

Motivation: Copy is slow.

Let's look at the other assignment:

```
s = a;
```

- **Copy** is necessary here, because `a` lives long. It is not destroyed immediately after this statement is executed.
- You cannot just "steal" the contents from `a`. The contents of `a` must be preserved.

Distinguish between the different kinds of assignments

```
s = a;
```

```
s = a + b;
```

What is the key difference between them?

- `s = a` is an assignment from an **lvalue**,
- while `s = a + b` is an assignment from an **rvalue**.

If we only have the copy assignment operator, there is no way of distinguishing them.

*** Define two different assignment operators, one accepting an lvalue and the other accepting an rvalue?**

Rvalue References

A kind of reference that is bound to **rvalues**:

```cpp
int &r = 42;               // Error: Lvalue reference cannot be bound to rvalue.
int &&rr = 42;             // Correct: `rr` is an rvalue reference.
const int &cr = 42;        // Also correct:
                           // Lvalue reference-to-const can be bound to rvalue.
const int &&crr = 42;      // Correct, but useless:
                           // Rvalue reference-to-const is seldom used.

int i = 42;
int &&rr2 = i;             // Error: Rvalue reference cannot be bound to lvalue.
int &r2 = i * 42;          // Error: Lvalue reference cannot be bound to rvalue.
const int &cr2 = i * 42; // Correct
int &&rr3 = i * 42;        // Correct
```

- Lvalue references (to non-`const`) can only be bound to lvalues.
- Rvalue references can only be bound to rvalues.

Overload Resolution

Such overloading is allowed:

```
void fun(const std::string &);
void fun(std::string &&);
```

- `fun(s1 + s2)` matches `fun(std::string &&)`, because `s1 + s2` is an rvalue.
- `fun(s)` matches `fun(const std::string &)`, because `s` is an lvalue.
- Note that if `fun(std::string &&)` does not exist, `fun(s1 + s2)` also matches `fun(const std::string &)`.

We will see how this kind of overloading benefit us soon.

Move Operations

Overview

The **move constructor** and the **move assignment operator**.

```
struct Widget {
  Widget(Widget &&) noexcept;
  Widget &operator=(Widget &&) noexcept;
  // Compared to the copy constructor and the copy assignment operator:
  Widget(const Widget &);
  Widget &operator=(const Widget &);
};
```

- Parameter type is **rvalue reference**, instead of lvalue reference-to-`const`.
- **`noexcept` is (almost always) necessary!** $\Rightarrow$ Think about why.

The Move Constructor

Take the `Dynarray` as an example.

```
class Dynarray {
  int *m_storage;
  std::size_t m_length;
public:
  Dynarray(const Dynarray &other) // copy constructor
    : m_storage(new int[other.m_length]), m_length(other.m_length) {
    for (std::size_t i = 0; i != m_length; ++i)
      m_storage[i] = other.m_storage[i];
  }
  Dynarray(Dynarray &&other) noexcept // move constructor
    : m_storage(other.m_storage), m_length(other.m_length) {
    other.m_storage = nullptr;
    other.m_length = 0;
  }
};
```

The Move Constructor

```cpp
class Dynarray {
  int *m_storage;
  std::size_t m_length;
public:
  Dynarray(Dynarray &&other) noexcept // move constructor
    : m_storage(other.m_storage), m_length(other.m_length) {


  }
};
```

1. *Steal* the resources of `other`, instead of making a copy.

The Move Constructor

```cpp
class Dynarray {
  int *m_storage;
  std::size_t m_length;
public:
  Dynarray(Dynarray &&other) noexcept // move constructor
    : m_storage(other.m_storage), m_length(other.m_length) {
    other.m_storage = nullptr;
    other.m_length = 0;
  }
};
```

1. *Steal* the resources of `other`, instead of making a copy.
2. Make sure `other` is in a valid state, so that it can be safely destroyed.

* **Take ownership of `other`'s resources!**

The Move Assignment Operator

Take ownership of `other` 's resources!

```
class Dynarray {
public:
  Dynarray &operator=(Dynarray &&other) noexcept {


      m_storage = other.m_storage; m_length = other.m_length;


    return *this;
  }
};
```

1. *Steal* the resources from `other` .

The Move Assignment Operator

```
class Dynarray {
public:
  Dynarray &operator=(Dynarray &&other) noexcept {


      m_storage = other.m_storage; m_length = other.m_length;
      other.m_storage = nullptr; other.m_length = 0;

    return *this;
  }
};
```

1. *Steal* the resources from `other`.
2. Make sure `other` is in a valid state, so that it can be safely destroyed.

Are we done?

The Move Assignment Operator

```
class Dynarray {
public:
  Dynarray &operator=(Dynarray &&other) noexcept {

      delete[] m_storage;
      m_storage = other.m_storage; m_length = other.m_length;
      other.m_storage = nullptr; other.m_length = 0;

    return *this;
  }
};
```

0. **Avoid memory leaks!**
1. *Steal* the resources from `other`.
2. Make sure `other` is in a valid state, so that it can be safely destroyed.

Are we done?

The Move Assignment Operator

```cpp
class Dynarray {
public:
  Dynarray &operator=(Dynarray &&other) noexcept {
    if (this != &other) {
      delete[] m_storage;
      m_storage = other.m_storage; m_length = other.m_length;
      other.m_storage = nullptr; other.m_length = 0;
    }
    return *this;
  }
};
```

0. **Avoid memory leaks!**
1. *Steal* the resources from `other`.
2. Make sure `other` is in a valid state, so that it can be safely destroyed.

* **Self-assignment safe!**

Lvalues are Copied; Rvalues are Moved

Before we move on, let's define a function for demonstration.

Suppose we have a function that concatenates two `Dynarray` s:

```
Dynarray concat(const Dynarray &a, const Dynarray &b) {
  Dynarray result(a.size() + b.size());
  for (std::size_t i = 0; i != a.size(); ++i)
    result.at(i) = a.at(i);
  for (std::size_t i = 0; i != b.size(); ++i)
    result.at(a.size() + i) = b.at(i);
  return result;
}
```

Which assignment operator should be called?

```
a = concat(b, c);
```

Lvalues are Copied; Rvalues are Moved

Lvalues are copied; rvalues are moved ...

```
a = concat(b, c); // calls move assignment operator,
                  // because `concat(b, c)` is an rvalue.
a = b; // calls copy assignment operator
```

Lvalues are Copied; Rvalues are Moved

Lvalues are copied; rvalues are moved ...

```
a = concat(b, c); // calls move assignment operator,
                  // because `concat(b, c)` generates an rvalue.
a = b; // copy assignment operator
```

... but rvalues are copied if there is no move operation.

```
// If Dynarray has no move assignment operator, this is a copy assignment.
a = concat(b, c)
```

Synthesized Move Operations

Like copy operations, we can use `=default` to require a synthesized move operation that has the default behaviors.

```
struct X {
  X(X &&) = default;
  X &operator=(X &&) = default;
};
```

- The synthesized move operations perform member-wise initialization/assignment of the data members, in their declaration order, from an *xvalue* [8] argument.
  - i.e. `X(X&& o) : mem1(std::move(o.mem1)), mem2(std::move(o.mem2)), ... {}`.
- The synthesized move operations are `noexcept` if all the operations involved are `noexcept`.

Move operations can also be deleted by `=delete`, but be careful ... [9]

The Rule of Five: Idea

The updated *copy control members*:

- copy constructor
- copy assignment operator
- move constructor
- move assignment operator
- destructor

If one of them has a user-provided version, the copy control of the class is thought of to have special behaviors. (Recall "the rule of three".)

The Rule of Five: Rules

- The move constructor or the move assignment operator will not be generated if any of the rest four members have a user-declared version.
- The copy constructor or copy assignment operator, if not provided by the user, will be implicitly `delete`d if the class has a user-provided move operation.
- The generation of the copy constructor or copy assignment operator is **deprecated** (since C++11) when the class has a user-declared copy operation or a destructor.
  - This is why some of you see this error:

    ```
    Implicitly-declared copy assignment operator is deprecated, because the
    class has a user-provided copy constructor.
    ```

The Rule of Five

The *copy control members* in modern C++:

- copy constructor
- copy assignment operator
- move constructor
- move assignment operator
- destructor

The Rule of Five: Define zero or five of them.

How to Invoke a Move Operation?

Suppose we give our `Dynarray` a label:

```
class Dynarray {
  int *m_storage;
  std::size_t m_length;
  std::string m_label;
};
```

The move assignment operator should invoke the **move assignment operator** on `m_label`. But how?

```
m_label = other.m_label; // calls copy assignment operator,
                         // because `other.m_label` is an lvalue.
```

`std::move`

`std::move`

Defined in `<utility>`

`std::move(x)` performs an **lvalue to rvalue cast**:

```
int ival = 42;
int &&rref = ival; // Error
int &&rref2 = std::move(ival); // Correct
```

Calling `std::move(x)` tells the compiler that:

- `x` is an lvalue, but
- we want to treat `x` as an **rvalue**.

`std::move`

`std::move(x)` indicates that we want to treat `x` as an **rvalue**, which means that `x` will be *moved from*.

The call to `std::move` **promises** that we do not intend to use `x` again,

- except to assign to it or to destroy it.

A call to `std::move` is usually followed by a call to some function that moves the object, after which **we cannot make any assumptions about the value of the moved-from object.**

```
void foo(X &&x);       // moves `x`
void foo(const X &x); // copies `x`
foo(std::move(x)); // matches `foo(X&&)`, so that `x` is moved.
```

"`std::move` does not *move* anything. It just makes a *promise*."

Use `std::move`

Suppose we give every `Dynarray` a special "label", which is a string.

```cpp
class Dynarray {
  int *m_storage;
  std::size_t m_length;
  std::string m_label;
public:
  Dynarray(Dynarray &&other) noexcept
      : m_storage(other.m_storage), m_length(other.m_length),
        m_label(std::move(other.m_label)) { // not self-assignment safe!!
    other.m_storage = nullptr;
    other.m_length = 0;
  }
};
```

The standard library facilities ought to define efficient and correct move operations.

Use `std::move`: self-assignment safe

Suppose we give every `Dynarray` a special "label", which is a string.

```
class Dynarray {
  int *m_storage;
  std::size_t m_length;
  std::string m_label;
public:
  Dynarray &operator=(Dynarray &&other) noexcept {
    if (this != &other) {
      delete[] m_storage;
      m_storage = other.m_storage; m_length = other.m_length;
      m_label = std::move(other.m_label);
      other.m_storage = nullptr; other.m_length = 0;
    }
    return *this;
  }
};
```

Use `std::move`

Why do we need `std::move`?

```
class Dynarray {
public:
  Dynarray(Dynarray &&other) noexcept
      : m_storage(other.m_storage), m_length(other.m_length),
        m_label(other.m_label) { // Isn't this correct?
    other.m_storage = nullptr;
    other.m_length = 0;
  }
};
```

`other` is an rvalue reference, so ... ?

An rvalue reference is an lvalue.

`other` is an rvalue reference, **which is an lvalue**.

- To move the object that the rvalue reference is bound to, we must call `std::move`.

```
class Dynarray {
public:
  Dynarray(Dynarray &&other) noexcept
      : m_storage(other.m_storage), m_length(other.m_length),
        m_label(other.m_label) { // `other.m_label` is copied, not moved.
    other.m_storage = nullptr;
    other.m_length = 0;
  }
};
```

An rvalue reference is an lvalue! Does that make sense?

Lvalues persist; Rvalues are ephemeral.

The lifetime of rvalues is often very short, compared to that of lvalues.

- Lvalues have persistent state, whereas rvalues are either **literals** or **temporary objects** created in the course of evaluating expressions.

An rvalue reference **extends** the lifetime of the rvalue that it is bound to.

```
std::string s1 = something(), s2 = some_other_thing();
std::string &&rr = s1 + s2; // The state of the temporary object is "captured"
                            // by the rvalue reference, without which the
                            // temporary object will be destroyed.
std::cout << rr << '\n'; // Now we can use `rr` just like a normal string.
```

Golden rule: **Anything that has a name is an lvalue.**

- The rvalue reference has a name, so it is an lvalue.