Summary

Members of a class

- A class can have data members and member functions.
- Access control: `private`, `public`.
  - One difference between `class` and `struct`: Default access.
- The `this` pointer: has type `X *` (`const X *` in `const` member functions). It points to the object on which the member function is called.
- `const` member function: guarantees that no modification will happen.

Summary

The followings hold for **all constructors**, no matter how they are defined:

- A constructor initializes **all** data members in order in which they are declared.
- The initialization of **all** data members is done before the function body of a constructor is executed.

In a constructor, a member is initialized as follows:

- If there is an initializer for it in the initializer list, use it.
- Otherwise, if it has an in-class initializer, use it.
- Otherwise, it is default-initialized. If it cannot be default-initialized, it leads to a compile-error.

Summary

Default constructors

- The default constructor defines the behavior of default-initialization.
- The default constructor is the constructor with an empty parameter list.
- If we have not defined **any constructor**, the compiler will try to synthesize a **default constructor** as if it were defined as `ClassName() {}`.
    - The compiler may fail to do that if some member has no in-class initializer and is not default-initializable. In that case, the compiler gives up (without giving an error).
- We can use `= default` to ask for a synthesized default constructor explicitly.

Summary

Lifetime of an object:

- depends on its **storage**: local non-`static`, global, allocated, ...
- **Initialization** marks the beginning of the lifetime of an object.
  - Classes can control the way of initialization using **constructors**.
- When the lifetime of an object ends, it is **destroyed**.
  - If it is an object of class type, its **destructor** is called right before it is destroyed.

Summary

Copy control

- Usually, the **copy control members** refer to the copy constructor, the copy assignment operator and the destructor.
- Copy constructor: `ClassName(const ClassName &)`
- Copy assignment operator: `ClassName &operator=(const ClassName &)`
  - It needs to be **self-assignment safe**.
- Destructor: `~ClassName()`
- `=default`, `=delete`
- The rule of three.

Summary

Type alias members

- Type alias members belong to the class, not individual objects, so they are accessed via `ClassName::AliasName`.
- The class can controls the accessibility of type alias members.

`static` members

- `static` data members are like global variables, but in the class's scope.
- `static` member functions are like normal non-member functions, but in the class's scope. There is no `this` pointer in a `static` member function.
- A `static` member belongs to the class, instead of any individual object.

Summary

`friend`

- A `friend` declaration allows a function or class to access private (and protected) members of another class.
- A `friend` is not a member.

Definitions and declarations

- A class definition includes declarations of all its members.
- A member function can be declared in the class body and then defined outside.
- A class type is an incomplete type if only its declaration (without a definition) is present.

Summary

Destructors

- Destructors are called automatically when an object's lifetime ends. They often do some clean up.
- The members are destroyed **after** the function body is executed. They are destroyed in reverse order in which they are declared.
- The compiler generates a destructor (in most cases) if none is provided. It just destroys all its members.

Summary

Rvalue references

- are bound to rvalues, and extends the lifetime of the rvalue.
- Functions accepting `X &&` and `const X &` can be overloaded.
- An rvalue reference is an lvalue.

Move operations

- take ownership of resources from the other object.
- After a move operation, the moved-from object should be in a valid state that can be safely assigned to or destroyed.
- `=default`
- The rule of five: Define zero or five of the special member functions.

Summary

`std::move`

- does not move anything. It only performs an lvalue-to-rvalue cast.
- `std::move(x)` makes a promise that `x` can be safely moved from.

In modern C++, unnecessary copies are greatly avoided by:

- copy-elision, which avoids the move or copy of temporary objects, and
- move, with the `return`ed lvalue treated as an rvalue, and

Notes

[1] More precisely, every *non-* `static` member belongs to an object.

[2] A `const` member function cannot modify its data members, unless that member is marked `mutable`.

[3] A constructor does not have a return type according to the standard. But it behaves as if its return type is `void`. Some compilers (such as Clang) may also treat it as if it returns `void`.

[4] In-class initializers cannot be provided in the form `(...)`. The parentheses here will be treated as part of a function declaration.

Notes

[5] Objects created by `new` / `new[]` are not required to destroyed. A `delete` / `delete[]` expression will destroy it, but it is not mandatory. So you can still create an object with a deleted destructor (see 7) by a `new` expression, but you can't `delete` it, which possibly leads to memory leak.

[6] A class can have many prospective destructors since C++20.

[7] If no user-declared destructor is provided for a class type, the compiler will always **declare** a destructor as an `inline` `public` member of its class.

If an implicitly-declared destructor is not deleted, it is **implicitly-defined** by the compiler when it is **odr-used**. In some very special cases the compiler may fail to define the destructor (e.g. due to a member whose destructor is inaccessible). In that case, the destructor is implicitly deleted.

Notes

[8] An xvalue is an rvalue that has an identity. A typical xvalue is one obtained from `std::move` applied to an lvalue.

[9] We seldom delete move operations. In most cases, we want rvalues to be copied if move is not possible. An explicitly deleted move operation will make rvalues not copyable, because deleted functions still participate in overload resolution.

An implicitly deleted move operation will be ignored in overload resolution. This is addressed by the defect report CWG 1402 and is applied retroactively to C++11. Note that this change of behavior did not come into effect when the book *C++ Primer, 5e* was published.