

CS100 Lecture 16

Classes III

Contents

- Type alias members
- `static` members
- `friend`
- Definition and declaration
- Destructors revisited

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.

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

Usage outside the class: `ClassName::TypeAliasName`

```
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(); // s is an object of `std::string`  
std::vector<int>::size_type j = v.size(); // v is an object of `vector<int>`  
std::list<int>::size_type k = l.size(); // l is an object of `list<int>`
```

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.
- Define `Container::size_type` to achieve good **consistency** and **generality**.

static members

static data members

A static data member:

```
class A {  
    public:  
        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 using `private`.

static data members

A static data member:

```
class A {  
    public:  
        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::` to access `something` outside the class.

static data members

A static data member:

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

`something` 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.

static data members: Example

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

```
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

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

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

```
class Y {
public:
    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.

```
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

```
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.

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

Or initialize it with some value explicitly:

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


static data members

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

```
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 using `private`.

static member functions

A static member function:

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

A::fun 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 pointed to by a this pointer.

A::fun has no way of accessing any non-static member.

friend

friend functions

Recall the `Student` class:

```
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::stoi(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

A *non-member* function:

```
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 class member.

friend functions

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

```
class Student {  
    friend void print(const 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::stoi(id.substr(0, 4))) {}  
    auto graduated(int year) const { return year - m_entranceYear >= 4; }  
    // ...  
};
```

friend functions

Add a `friend` declaration.

```
class Student {  
    friend void print(const Student &);  
    // ...  
};
```

A `friend` is **not** a member! You can put this `friend` declaration **anywhere in the class body**. The access specifiers 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:

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

```
class Widget {  
    int m_handle;  
    int m_length;  
    std::vector<int> m_gadgets;  
  
public:  
    Widget();  
    Widget(int, int);  
    void set_handle(int);  
    const std::vector<int> &get_gadgets() const;  
};
```

Define a member function outside the class body

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

```
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;
```

If we only see the **declaration** of a class, we have no knowledge about its members, 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 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

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

- For global and `static` objects, on termination of the program.
- For local non-`static` 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**, such as releasing the resources the object owns.

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;  
    void setName(const std::string &);  
    void print() const;  
    bool graduated(int) 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 a `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.

What does `Student::~~Student` need to do? Does a `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.

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

Destructors

```
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 data members.

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

- After the function body is executed, **all the data members** are destroyed automatically, **in reverse order** in which they are declared.
 - For members of class types, 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.
- 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 if there is no user-declared destructor.

- The generated destructor is `public` by default.
- The generated destructor has an empty function body `{}`.
- It does nothing but destroy the data members.

We can explicitly ask for one by writing `= default;`.

Summary

Type alias members

- Type alias members belong to the class, not individual objects, so they are accessed via `ClassName::`.
- 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.

Summary

friend

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

Definition and declaration

- 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

- A destructor is called automatically when an object's lifetime ends. They often do some cleanup.
- The data 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 the data members.

Notes

¹ Objects created by `new` / `new[]` are not required to be destroyed. A `delete` / `delete[]` expression will destroy it, but it is not mandatory. So you can still create an object with a deleted destructor by a `new` expression, but you can't `delete` it, which possibly leads to memory leak.