CS100 Recitation 8

An intro to `class`

*Warm Up

Idea: Search the Internet for things that you do not know (the following code is a JavaScript try-catch block).

```
try {
  // do something
} catch (error) {
  location.href = "https://www.google.com/search?q=${error}"
}
```

Contents

Object-Oriented Programming (OOP)

面向对象编程

https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/

Motivations

- To implement real-world entities like inheritance, hiding, polymorphism, etc. in programming.
- To bind together the **data** and the **functions** that operate on them so that no other part of the code can access this data except that function.

Basic Concepts

- *Class and Object*（类与对象）.
- *Encapsulation* (封装).
- Inheritance.
- Polymorphism.
- Abstraction.

Classes and Objects

类与对象

Class

- A **blueprint** representing a group of objects which shares some common properties and behaviors.
- A user-defined **data type**, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

Object

- An **instance** of a Class.
- When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Access Members of A Class

Member access

- Use `.` or `->`. (link)

Access control

- `private` members: Only accessible to code inside the class and `friend`s.
- `public` members: Accessible to all parts of the program.
- `protected` members ...

Access control

```
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`.

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`.
- They are widely used to avoid name conflicts:

```
void classx::set_x(int x) {this->x = x;}
```

The `this` pointer: a debate

Q: Please look at the following two scripts that do the same task, one with `this` and the other isn't. So when should we use `this`?

```
void Classx::memberfunction()
{
    this->doSomething();
    std::cout << this->memberVar;
}
```

```
void Classx::memberfunction()
{
    doSomething();
    std::cout << memberVar;
}
```

The `this` pointer: a debate

- Generally, there isn't a clear answer for this quesiton. That is, you can use `this` explicitly or follow a certain **name convention rule** to avoid using `this`.
- For example, some programmers suffix the member variables with a single underscore `_` to avoid possible name conflicts (e.g. use `m_` instead of `m`).
- However, for certain cases in templated codes, we have got to use `this`.
    - Example 1.
    - Example 2.

Constructors

Constructors (构造函数)

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

```
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
```

Constructors

1. The constructor name is the class name.
2. The constructor **does not** have a return type, it can contain a return, but it cannot return a value.
3. When we call the constructor, we are creating an instance of the class.
4. The constructor is responsible for initializing the object, including the initialization of **all members**.

Member Initialization Methods

From the most specific to the most general:

1. If the member appears in the constructor's **initializer list**, it is initialized using the initializer provided in the list.
2. Otherwise, if the member has an **in-class initializer**, that initializer is used.
3. Otherwise, if it can be **default-initialized**, it will be default-initialized.
4. Otherwise, the member **cannot** be initialized in the current constructor:
    - If it is a user-defined constructor, a compile-time error will occur.
    - If it is a compiler-generated constructor, the constructor is a deleted function.

Constructor initializer list (初始值列表)

- The initialization of all members is completed **before entering the function body**, and their initialization methods are partly determined by the constructor initializer list:

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

- 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.

In-class initializer (類内初始值)

We can assign values to member variables during declaration.

```
struct Point2d {
  double x;
  double y = 0;
  Point2d() = default;
  Point2d(double x_) : x(x_) {} // y = 0
};
Point2d p; // value of x is undefined and y = 0
Point2d p2(3.14); // x = 3.14, y = 0
```

If a member does not appear in a constructor's initialization list, compiler will use the in-class initializer instead of the default initialization.

In-class initializer (类内初始值)

We can't use `()` here, and use `{}` instead.

```
struct X {
  // 错误：受限于编译器的设计，它会在语法分析阶段被认为是函数声明
  std::vector<int> v(10);

  // 这毫无疑问是声明一个函数，而非设定类内初始值
  std::string s();

};
```

Default Constructor (默认构造函数)

- A special constructor that does not accept parameters.
- It is specifically used for default initialization of objects, as no parameters are needed when calling it, effectively not requiring any initializers.
- Value initialization of **class types** (almost) always involves calling the default constructor.

```
class Point2d {
  double x, y;
public:
  Point2d() : x(0), y(0) {}
  Point2d(double x_, double y_) : x(x_), y(y_) {}
};
Point2d p1;       // Calls the default constructor, (0, 0)
Point2d p2{};     // Value-initialization, calls the default constructor, (0, 0)
Point2d p3(3, 4); // Calls Point2d(double, double), (3, 4)
Point2d p4();     // This is a function
```

Default Constructor (默认构造函数)

- If the class has a user-declared constructor, the compiler will **not** generate a default constructor.
- If no other constructors are defined, or if we explicitly request the compiler to generate a default constructor using `= default`:
  - The compiler will synthesize a default constructor with default behavior.
  - "Default behavior": It initializes the members one by one in the order of their declaration.
    - For members with in-class initializers, the in-class initializer will be used.
    - For other members, they will be default-initialized.

Q: Why do we need a constructor initializer list?

Consider the following case:

```
class Student {
    const std::string name_;
    std::vector<int>& classIDs_;
public:
    Student(const std::string& name, std::vector<int>& classIDs)
    {
        name_ = name;
        classIDs_ = classIDs;
    }
};
```

Q: Why do we need a constructor initializer list?

1. For references and `const` objects, we need an initializer list.
2. Moreover, if a data member is default-initialized and then assigned when could have been initialized directly, it may lead to low efficiency.

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

Q: Is a default constructor needed?

[Best practice] When in doubt, leave it out. If the class does not have a "default state", it should not have a default constructor!

- Do not define one arbitrarily or letting it `= default`. This leads to pitfalls.
- Calling the default constructor of something that has no "default state" should result in a **compile error**, instead of being allowed arbitrarily.

Copy Constructor

https://en.cppreference.com/w/cpp/language/copy_constructor

Copy constructor

To construct an object from an already exist object (like clone an object).

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 that the `=` in `Type b = a;` **is not an assignment operator**.

Copy constructor

1. We can define our own copy constructor

```
className::className(const className& other) {...}
```

2. If a class does not have a user-declared copy constructor, the compiler will try to synthesize one. And we can explicitly ask for one:

```
className::className(const className&) = default;
```

3. If we need to avoid construction from copying, try:

```
className::className(const className&) = delete;
```

Copy-assignment Operator

https://en.cppreference.com/w/cpp/language/copy_assignment

Copy-assignment

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

```
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.

Copy-assignment operator

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

- `a = b` is equivalent to `a.operator=(b)`.
- A general form is:

```
className& operator=(const className&) {...}
```

- 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`.

Copy-assignment operator

What if we make the return type of `operator=` to `className` instead of `className&`?

Try this out:

```
class A {
public:
    A() {std::cout << "constructor\n";};
    A(const A& other) {std::cout << "copy-constructor\n";}
    A operator=(const A& other) {std::cout << "operator=\n"; return *this;}
};
int main() {
    A a1;
    A a2 = a1;
    A a3{a1};
    std::cout << "<<< construction complete >>>\n";
    a3 = a2;
    return 0;
}
```

Notes

1. Please be aware of what you want to do with copy-assignment.
2. Assignment operators should be **self-assignment-safe**.
3. Pay attention to the return type (`T&` is favored in order to allow chaining asssignments).

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;`.

Destructors

https://en.cppreference.com/w/cpp/language/destructor

Destructors

- When an object is destroyed, it must deallocate the memory.
- A destructor of a class is the member function that is **automatically** called when an object of that class type is destroyed.
- If the object owns some resources (e.g., dynamic memory), destructors can be made use of to avoid leaking.
- 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.
- A general form would be:

```
~className() {...};
```

Exercise

Finish the destructor in `linkedList.cpp`.

Definition and Declaration

Class definition

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

```
class Demo {
    int val;
public:
    Demo() : val(0) {};
    int get_val() const; // A declaration only
};
```

We can define member functions outside with `::` :

```
int Demo::get_val() const {return val;}
```

In practice, we store the definition of a class in a `.h` file while implement the definitions of all member functions in another `.cpp` file.

A Question

Notice that in previous slides I mentioned *Encapsulation*, does anyone have some thoughts about this concept after this class?