

# CS100 Lecture 14

## Class Basics I

# Contents

## Class basics

- Members of a class
  - Access
  - The `this` pointer
- Constructors
  - Constructor initializer list
  - Default constructors

# Members of a class

# A simple `class`

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

```
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 <sup>1</sup> 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

```
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` s in later lectures.
- `public` members: Accessible to all parts of the program.

# 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;  
};
```

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

```
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++.



## The `this` pointer

```
class Student {  
    // ...  
public:  
    bool graduated(int year) const;  
};  
  
Student s = someValue();  
if (s.graduated(2023))  
    // ...
```

How many parameters does `graduated` have?

# The `this` pointer

```
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!

# The `this` pointer

```
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:

```
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");
```

# 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 <sup>2</sup>.
- 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.

# 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.

```
class Student {  
public:  
    void foo() 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:

```
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:

```
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**.



## const member functions

```
class Student {  
    std::string name, id;  
    int entranceYear;  
public:  
    const std::string &getName() const { return name; }  
    const std::string &getID() const { return id; }  
    bool valid() const { return id.substr(0, 4) == std::to_string(entranceYear); }  
    void adjustID() { id = std::to_string(entranceYear) + id.substr(4); }  
};
```

`str.substr(pos, len)` returns the substring of `str` starting from the position indexed `pos` with length `len`.

- If `len` is not provided, it returns the **suffix** starting from the position indexed `pos`.

# Constructors

Often abbreviated as "ctors".

# Constructors

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  
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`<sup>3</sup>). 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**:

```
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 `= ...`.<sup>4</sup>

```
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:

```
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:

```
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] Always use an initializer list in a constructor.

- 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.

```
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.

```
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`.

## Is a default constructor needed?

First, if you need to use arrays, you almost certainly need a default constructor:

```
Student s[1000]; // All elements are default-initialized
                // by the default constructor.
Student s2[1000] = {a, b}; // The first two elements are initialized to
                          // `a` and `b`. The rest are initialized by the
                          // default constructor.
```

A `std::vector` does not require that:

```
// In this code, the default constructor of `Student` is not called.
std::vector<Student> students;
for (auto i = 0; i != n; ++i)
    students.push_back(some_student());
```



## Is a default constructor needed?

If a class has no user-declared constructors, the compiler will try to synthesize a default constructor.

```
class X {}; // No user-declared constructors.  
X x; // OK: calls the compiler-synthesized default constructor
```

The synthesized default constructor initializes the data members as follows:

- If a data member has an in-class initializer, it is initialized according to the in-class initializer.
- Otherwise, default-initialize that member. If it cannot be default-initialized, the compiler will give up -- no default constructor is generated.

## Is a default constructor needed?

If a class has any user-declared constructors but no default constructor, the compiler **will not** synthesize a default constructor.

You may ask for a default constructor with `= default;`:

```
class Student {  
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() = default;  
};
```

## Is a default constructor needed?

It depends on the **design**:

- If the class has a default constructor, what should be the behavior of it? Is there a reasonable "default state" for your class type?

For `Student`: What is a "default student"?

## Is a default constructor needed?

It depends on the **design**:

- If the class has a default constructor, what should be the behavior of it? Is there a reasonable "default state" for your class type?

For `Student`: What is a "default student"?

- There seems to be no such thing as a "default student" (in a normal design). Therefore, `Student` should not have a default constructor.

## 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.

# 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.



# Notes

- <sup>1</sup> Every *non-static* member belongs to an object. All data members mentioned in the slides of this lecture are *non-static*.
- <sup>2</sup> A *const* member function cannot modify its data members, unless that member is marked *mutable*.
- <sup>3</sup> 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*.
- <sup>4</sup> In-class initializers cannot be provided in the form *(...)*. The parentheses here will be treated as part of a function declaration.