

CS100 Recitation 7

GKxx

April 4, 2022

Drawbacks of a Simple struct

Take the `Linked_list` as an example:

- Users can directly access and modify the structure of the list, **without letting the list know!**
- Even though methods of 'create' and 'destroy' are provided, memory management is still a problem because users may forget to call them (or fail to call them correctly).
- The name of every function starts with 'linked_list', which is lengthy and inconvenient.



Separate Implementation Details and Interfaces

```
struct Point2d {  
    private:  
        // implementation details  
        double x, y;  
    public:  
        // interfaces  
        void set_x(double new_x)  
            { x = new_x; }  
        void set_y(double new_y)  
            { y = new_y; }  
        double get_x()  
            { return x; }  
        double get_y()  
            { return y; }  
};
```

Access modifiers:

- **private**: Only the code inside the class (or in a friend) can access.
- **public**: Everyone can access.
- **protected**: Only the code inside the class or in a subclass, or in a friend can access.

Separate Implementation Details and Interfaces

- Implementation details should be invisible to others.
- Interfaces are defined for others to use.

```
// In C++, we can directly use the name Point2d without the  
    struct keyword.
```

```
// The weird typedefs are not needed, either.
```

```
Point2d p;
```

```
p.x = 4.2; // Error!
```

```
p.set_x(4.2);
```

```
p.set_y(3.5);
```

```
std::cout << "(" << p.get_x() << ", "  
           << p.get_y() << ")" << std::endl;
```

class or struct ?

In C++, **the only differences** between `class` and `struct` are

- Default access level for a `class` is `private`, while for a `struct` is `public`.

```
class Point {  
    double x, y; // private here  
    // Other members.  
};
```

```
struct Point {  
    double x, y; // public here  
    // Other members.  
};
```

- Default inheritance protection level for a `class` is `private`, while for a `struct` is `public`.

const Member Functions

```
void print_point(const Point2d &p) {  
    std::cout << "(" << p.get_x() << ", "  
               << p.get_y() << ")" << std::endl;  
}
```

- The parameter should be declared as `const` reference, since it is not modified.
- However, the code above won't compile.
- We need to specify what we can do on `const` objects.

const Member functions

```
struct Point2d {  
    private:  
        double x, y;  
    public:  
        void set_x(double new_x)  
            { x = new_x; }  
        void set_y(double new_y)  
            { y = new_y; }  
        double get_x() const  
            { return x; }  
        double get_y() const  
            { return y; }  
};
```

- On a non-`const` object, both `const` members and non-`const` members can be called.
- On a `const` object, only the `const` members can be called.
- A `const` member function should not modify the data members.

The `this` Pointer

Inside a member function, when we refer to the name of a member, we are in fact referring to it through the `this` pointer.

```
class Point2d {  
    double x, y;  
public:  
    void set_x(double new_x) {  
        this->x = new_x;    // equivalent to x = new_x;  
    }  
    // Other members.  
};
```

- `this` is a pointer that points to the object itself. For example, in 'class Point2d', `this` is of type `Point2d *`.

Name Lookup in class

An exception to the name lookup rule:

- Inside a `class`, all the class members are visible, no matter they are before or after the usage.

```
class Point2d {  
    public:  
        void set_x(double new_x)  
            { x = new_x; }           // OK: The member 'x' is visible here.  
        void set_y(double new_y)  
            { y = new_y; }  
        double get_x() const  
            { return x; }  
        double get_y() const  
            { return y; }  
    private:  
        double x, y;  
};
```

Defining Member Functions outside the class

A member function can be defined outside the `class` definition, **but must be declared inside the class**.

```
class Point2d {  
public:  
    void set_x(double new_x) {  
        x = new_x;  
    }  
    void set_y(double new_y) {  
        y = new_y;  
    }  
    double get_x() const;  
    double get_y() const;  
private:  
    double x, y;  
};
```

- Use `operator::` to refer to a name in the class scope.

```
double Point2d::get_x() const {  
    return x;  
}  
double Point2d::get_y() const {  
    return y;  
}
```

- The `const` keyword, if needed, must appear at both declaration and definition. It is a part of the function type.

Reference to the Object itself

```
class Point2d {  
    public:  
        Point2d &set_x(double new_x) {  
            x = new_x;  
            return *this;  
        }  
        Point2d &set_y(double new_y) {  
            y = new_y;  
            return *this;  
        }  
        // Other members.  
};
```

- `set_x` and `set_y` returns a reference to the object itself (which is an *lvalue*) by return `*this`;
- Then we can do:
`p.set_x(4.2).set_y(3.5);`

Constructors

Constructors ('ctors' for short) define the ways of initializing an object.

```
class Point2d {  
    public:  
        Point2d(double a, double b)  
            : x(a), y(b) {}  
        Point2d() : x(0), y(0) {}  
        // Other members.  
};
```

```
// Initializing a Point2d object  
        with x = 3.5, y = 4.2  
Point2d p1(3.5, 4.2);  
// Default-initialization calls  
        the default ctor with no  
        arguments.  
Point2d p2;  
// p2 is initialized with x = 0,  
        y = 0.
```

Constructors

- The name of a ctor is name of the class.
- The return-type is omitted, because a construction expression always returns the constructed object. The following statement will output 4.2.

```
std::cout << Point2d(4.2, 3.5).get_x() << std::endl;
```

- There is a **constructor initializer list** after the parameters, starting with ':', containing explicit initialization of **several** data members, separated with commas.
- The constructors are often **overloaded**, i.e. we may define several different ways of initialization.

The Ctor Initializer List

The most important part of a ctor is the **initializer list**.

- The ctor-init-list is the part where data members are initialized.
- **All the data members** will be initialized before the function body begins, **in the order in which they appear in the class definition**.
- If a data member does not appear in the ctor-init-list, it is **default-initialized**.
 - ▶ For built-in types, default-initialization makes the object hold an undefined value.
 - ▶ For class types, default-initialization is calling the **default-ctor** of that class.

Ctor-init-list: Examples

- The following ctor is **bad**: (It is default-initializing the members, and then assigning values to them.)

```
Point2d(double a, double b) {  
    x = a;  
    y = b;  
}
```

- The following ctor is **misleading**:

```
Point2d(double a, double b) : y(b), x(a) {}
```

- The following ctor first default-initializes `x`, and then initializes `y` with given value.

```
Point2d(double b) : y(b) {}
```


Ctor-init-list: Examples

A typical bug I made (years ago):

```
class Snake_game {  
    std::vector foods;  
    size_t num_food;  
    // Other members  
public:  
    Snake_game(size_t n, /* other args */)   
        : num_food(n), foods(num_food) {}  
    // Other members  
};
```

Importance of Using Ctor-init-list

It is strongly suggested to use ctor-init-lists **routinely**:

- Initialize the member directly, instead of first default-initialize it and then assign it with a value.
- Some members cannot be default-initialized or cannot be assigned.

```
class Foo {  
    const int x;  
    int y;  
public:  
    Foo(int a, int b) {  
        // Error: x cannot be assigned after initialization.  
        x = a;  
        y = b;  
    }  
};
```

Default Initialization

- The **default ctor** is the ctor with no arguments. It defines the behavior when the object is **default-initialized** or **value-initialized**.
- The following code outputs 'Liu Big God is so strong.'

```
class Point2d {  
    public:  
        Point2d() : x(0), y(0) {  
            std::cout << "Liu Big God is so strong.\n"  
        }  
        // Other members.  
};  
// in main  
Point2d p;
```

Default Initialization

```
class Line2d {  
    Point2d p0, v;  
public:  
    Line2d()  
        { std::cout << "Liu Big God is so powerful.\n" }  
    // Other members.  
};
```

What's the output of the following code (inside the main function)?

```
Line2d line;
```

```
Liu Big God is so strong.  
Liu Big God is so strong.  
Liu Big God is so powerful.
```

Have a Try

Define a class `Vector`:

- Define the data members `m_size`, `m_capacity` and `m_data`, denoting the number of elements stored, the maximum possible size of the current storage, and a pointer to the storage.
 - ▶ The prefix `m_` means 'member'.
- Define a default-ctor that initializes the `Vector` to an empty `Vector`, i.e. `m_size = 0`, `m_capacity = 0`, `m_data = nullptr`.
- Define a ctor that receives an unsigned number `n`, which initializes the `Vector` to be holding `n` value-initialized `ints`.
- Define a ctor that receives two pointers `begin` and `end` pointing to the beginning and the pass-end of an array. The ctor copies the values from the range of the array.

Constructors

```
class Vector {
    size_t m_size, m_capacity;
    int *m_data;
public:
    Vector() : m_size(0), m_capacity(0), m_data(nullptr) {}
    Vector(size_t n) : m_size(n), m_capacity(n),
        m_data(new int[n]()) {}
    Vector(int *begin, int *end)
        : m_size(end - begin), m_capacity(end - begin),
        m_data(new int[end - begin]()) {
        for (int *p = m_data; begin != end; ++begin, ++p)
            *p = *begin;
    }
};
```

nullptr

- In C, the null pointer NULL is defined to be of type `void *`, so that it can be converted to any pointer type.
- However, C++ does not allow implicit conversion from `void *` to other pointer types. It's possible that NULL is defined as `(long)0`.
 - ▶ C++ allows initialization of a pointer with integral literal 0, but not other values.
- Since C++11, we use `nullptr` as a well-defined null pointer. It is of type `std::nullptr_t`, a type defined in `<cstddef>`. (`stddef.h` does not have this!)

new and delete

- Roughly speaking, `new` and `delete` are like the C++ version of `malloc` and `free`: They allocate memory on heap.
- `new` not only **allocates the memory**, but also **constructs the object**.
- `delete` first **destructs the object**, and then **deallocates (frees) the memory**.
- `new` and `delete` will call the ctor and dtor of the type, while `malloc` and `free` won't!
- When you have some object on heap in C++, **NEVER** use `malloc/calloc/realloc/free`. You should always use `new` and `delete`.

new and delete

```
int *pi = new int;           // default initialization
int *pi2 = new int();        // value initialization
int *pia = new int[10];      // default initialization
int *pia2 = new int[10]();   // value initialization
int *pia3 = new int[10]{1, 2, 3}; // First three values: 1, 2, 3
                                // Others: value initialization

// call Point2d::Point2d(double, double)
Point2d *p = new Point2d(3, 4);
// call Point2d::Point2d()
Point2d *p2 = new Point2d;
// call Point2d::Point2d()
Point2d *p3 = new Point2d();
```

Initialization in `new`

- Whenever the explicit initializer is absent, **default initialization** is performed.
- Whenever an explicit initializer is provided but as an empty pair of parentheses, **value initialization** is performed.
- For a class type **with a user-defined default ctor**, both default initialization and value initialization call the default ctor.

The rules for initialization are **very complicated**. Now you only need to know these that are listed above.

new and delete

- To destroy the object created by `new` and free the memory, use `delete`.

```
int *pi = new int();  
// after some operations  
delete pi;
```

- For an array type, use `delete[]`.

```
int *pia = new int[10]();  
// after some operations  
delete[] pia;
```

- `delete` will call the destructor of a class type. (We will talk about destructors later.)

In-class_INITIALIZER

- An **in-class initializer** is used to define the default value for a member.
- When a member with an in-class initializer is **default initialized** or **value initialized**, it will be initialized with the in-class initializer.

```
class Point2d {  
    double x = 0, y = 0;  
    // Other members.  
};
```

```
class Line2d {  
    // Curly braces are allowed  
    here since C++11.  
    Point2d p0{0, 0}, v{1, 1};  
    // Other members.  
};
```

In-class_INITIALIZER

Due to the limitation of compilers (you will learn about it in CS131), parentheses are not allowed in in-class initializer: They will be viewed as function declarations!

```
class Line2d {  
    // Error: They are seen as function declarations.  
    Point2d p0(0, 0), v(1, 1);  
    int foo(); // Exactly a member function declaration.  
    // Other members.  
};
```

Synthesized Default Ctor

The compiler will generate a default ctor if the following conditions are satisfied:

- Each member without a in-class initializer is **default-initializable**, and
- There is no user-defined ctors, or the default ctor is defined as `=default`.

Many kinds of members are not **default-initializable**, e.g. a reference, or a class type member whose class does not have accessible default ctor.

Synthesized Default Ctor

The synthesized default ctor will

- initialize a member with the in-class initializer, if it has an in-class initializer.
- Otherwise, it will default-initialize the member.

Delegating Ctor

Since C++11, a ctor can **delegate** part or all of its work to another ctor:

```
class Point2d {  
    double x, y;  
public:  
    Point2d() : Point2d(0, 0) {  
        std::cout << "default ctor body\n";  
    }  
    Point2d(double a, double b) : x(a), y(b) {  
        std::cout << "Point2d(double, double) body\n";  
    }  
};  
// in main  
Point2d p;
```

The above code outputs “Point2d(double, double) body” first, and then “default ctor body”.

Example

```
class Vector {
    size_t m_size = 0, m_capacity = 0;
    int *m_data = nullptr;
public:
    // explicitly require a synthesized default ctor
    Vector() = default;
    Vector(size_t n) : m_size(n), m_capacity(n),
        m_data(new int[n]()) {}
    // delegate to Vector::Vector(size_t)
    Vector(int *begin, int *end) : Vector(end - begin) {
        for (int *p = m_data; begin != end; ++begin, ++p)
            *p = *begin;
    }
};
```

Destructor