智能指针

智能指针

C++11引入了3个智能类型的指针：

1. `std::unique_ptr<T>`：独占资源所有权的指针
2. `std::shared_ptr<T>`：共享资源所有权的指针
3. `std::weak_ptr<T>`：共享资源的观察者，需要和`std::shared_ptr<T>`一起使用，不影响资源的生命周期

std::unique_ptr

当我们独占资源的所有权时，我们可以使用`std::unique_ptr<T>`进行资源管理——离开`std::unique_ptr<T>`对象的作用域时，会自动释放资源

`std::unique_ptr<T>`的特性：

1. 使用`std::unique_ptr<T>`自动管理内存

自动释放内存 C++

```cpp
{
    std::unique_ptr<int> uptr = std::make_unique<int>(200);
    //...
    // 离开 uptr 的作用域的时候自动释放内存
}
```

2. `std::unique_ptr<T>`是move-only的

> 可以使用std::move或者std::make_unique为unique_ptr赋值

移动赋值 C++

```cpp
{
    std::unique_ptr<int> uptr = std::make_unique<int>(200);
    std::unique_ptr<int> uptr1 = uptr;  // 编译错误，std::unique_ptr<T> 是 move-only 的

    std::unique_ptr<int> uptr2 = std::move(uptr);
    assert(uptr == nullptr);
}
```

3. 自定义Deleter删除资源

a. 传递函数指针

C++

```cpp
void close_file(std::FILE* fp) {
    std::fclose(fp);
}
std::unique_ptr<std::FILE, void (*)(std::FILE*) > fp(
    std::fopen("demo.txt", "r"),
    close_file
);
```

b. 传递删除器

C++

```cpp
{
    struct FileCloser {
        void operator()(FILE* fp) const {
            if (fp != nullptr) {
                fclose(fp);
            }
        }
    };
    std::unique_ptr<FILE, FileCloser> uptr(fopen("test_file.txt", "w"));
}
```

c. 传递Lambda函数

```cpp
{
    std::unique_ptr<FILE, std::function<void(FILE*)>> uptr(
        fopen("test_file.txt", "w"),
        [](FILE* fp) {
            fclose(fp);
        });
}
// 其中std::function<void(FILE*)>类型定义可以被删除
```

4. 修改器
    a. `release`：返回一个指向被管理对象的指针，并释放管理权
    b. `reset`：替换被管理对象
    c. `swap`：交换被管理对象
5. 观察器
    a. `get`：返回指向被管理对象的指针
    b. `get_deleter`：返回用于析构被管理对象的删除器

std::shared_ptr

`std::shared_ptr<T>` 是通过指针保持对象共享所有权的智能指针，对资源做引用计数——当引用计数为0的时候，自动释放资源

```cpp
{
    std::shared_ptr<int> sptr = std::make_shared<int>(200);
    assert(sptr.use_count() == 1);  // 此时引用计数为 1
    {
        std::shared_ptr<int> sptr1 = sptr;
        assert(sptr.get() == sptr1.get());
        assert(sptr.use_count() == 2);   // sptr 和 sptr1 共享资源，引用计数为 2
    }
    assert(sptr.use_count() == 1);   // sptr1 已经释放
}
// use_count 为 0 时自动释放内存
```

1. 修改器

a. `reset`：替换被管理对象
b. `swap`：交换被管理对象

2. 观察器
   a. `get`：返回存储的指针
   b. `use_count`：返回shared_ptr所指向的对象的引用计数
   c. `unique`：检查所管理的对象是否仅由当前shared_ptr管理

std::weak_ptr

`std::weak_ptr` 要与 `std::shared_ptr` 一起使用

一个 `std::weak_ptr` 对象可以看作是 `std::shared_ptr` 对象管理的资源的观察者，它不影响共享资源的生命周期：

1. 如果需要使用 `std::weak_ptr` 正在观察的资源，可以将 `std::weak_ptr` 提升为 `std::shared_ptr`
2. 当 `std::shared_ptr` 资源被释放时， `std::weak_ptr` 会自动变成 `nullptr`

C++

```cpp
void Observe(std::weak_ptr<int> wptr) {
    if (auto sptr = wptr.lock()) {
        std::cout << "value: " << *sptr << std::endl;
    } else {
        std::cout << "wptr lock fail" << std::endl;
    }
}

std::weak_ptr<int> wptr;
{
    auto sptr = std::make_shared<int>(111);
    wptr = sptr;
    Observe(wptr);  // sptr 指向的资源没被释放，wptr 可以成功提升为 shared_ptr
}
Observe(wptr);  // sptr 指向的资源已被释放，wptr 无法提升为 shared_ptr
```

1. 修改器
   a. `reset`：替换被管理对象
   b. `swap`：交换被管理对象

2. 观察器
    a. `use_count`： 返回shared_ptr所指向的对象的引用计数
    b. `expired`： 查看被管理对象是否被删除
    c. `lock`： 创建管理被引用的对象的shared_ptr