



上海科技大学
ShanghaiTech University

CS100 Recitation 11

Yunxiang He

Modified From God GKxx



立志成才 报国裕民



Contents

- 拷贝控制实例：Message and Folder
- 多文件和动态链接





上海科技大学
ShanghaiTech University

拷贝控制实例



立志成才 报国裕民



Message and Folder

考虑一个**邮件系统**:

- 一个 Folder 中包含一些 Message
- 一个 Message 可能存在于多个 Folder 中, 但我们不希望拷贝 Message 的内容
它们具有各自的复制行为:
- 复制一个 Message: 复制其内容, 并且将新的 Message 也加入相应的 Folder
- 复制一个 Folder: 将它包含的所有 Message 也加入新的 Folder





Message and Folder

```
class Message {  
    std::string m_contents;  
    std::set<Folder *> m_folders;  
};  
class Folder {  
    std::set<Message *> m_messages;  
};
```

快速地插入、删除、查找元素的“集合”数据结构： `std::set` （标准库文件 `<set>` ）





.....欸，我超？

- 编译器报错了？

纳尼，情报是假的？



```
error: 'Folder' was not declared in this scope
  6 |     std::set<Folder *> m_folders;
    |               ^~~~~~
```

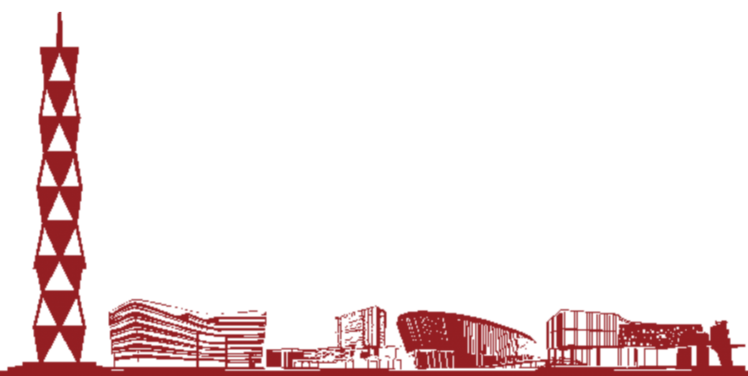
为什么?



上海科技大学
ShanghaiTech University



nmd为什么



立志成才报国裕民



Message and Folder

——循环依赖!

“循环依赖”的情形: Message 需要 Folder , Folder 需要 Message 。

- 无论谁在前,都会有一个名字查找发生失败。

幸好,这里**不完全类型就足够了**

- 我们只是使用了 Folder* 和 Message*
- 没有创建其对象,没有使用其成员

所以在前面补一个**声明**即可。





Message and Folder

```
class Folder; // declaration
class Message {
    std::string m_contents;
    std::set<Folder *> m_folders;
};
class Folder {
    std::set<Message *> m_messages;
};
```

先写几个简单的成员函数：添加一个 `message`，添加一个 `folder` 等等。





Message and Folder

Message

`s.insert(x)` 向集合 `s` 添加一个元素; `s.erase(x)` 从 `s` 中删除元素 `x`。

```
class Message {  
    std::string m_contents;  
    std::set<Folder *> m_folders;  
public:  
    void addFolder(Folder &folder) {  
        m_folders.insert(&folder);  
    }  
    void removeFolder(Folder &folder) {  
        m_folders.erase(&folder);  
    }  
};
```

```
class Folder {  
    std::set<Message *> m_messages;  
public:  
    void addMessage(Message &msg) {  
        m_messages.insert(&msg);  
    }  
    void removeMessage(Message &msg) {  
        m_messages.erase(&msg);  
    }  
};
```

有没有哪个地方应该使用 `const` ?





Message and Folder

```
class Message {  
    std::string m_contents;  
    std::set<Folder *> m_folders;  
public:  
    void addFolder(Folder &folder) {  
        m_folders.insert(&folder);  
    }  
    void removeFolder(Folder &folder) {  
        m_folders.erase(&folder);  
    }  
};
```

有没有哪个地方应该使用 `const` ?

- 表面上看, `addFolder` 和 `removeFolder` **不改变**被添加的 `folder`
- 但如果声明为 `const Folder &folder`, 就意味着 `m_folders` 里的元素也应该是 `const Folder *`。
- 而我们存储这些 `Folder` 的指针, 以后必然需要调用它们的 `non-const` 操作 (例如添加、删除信息)。
- 因此, `addFolder` 和 `removeFolder` 应该将那些不能修改的 `Folder` 拒之门外, 所以这里不能带 `const`。



Message and Folder

`Message` 的构造函数：接受一个字符串作为信息的内容。

```
class Message {  
    public:  
        // 拷贝左值, 移动右值  
        Message(std::string contents) : m_contents{std::move(contents)} {}  
};
```





Message and Folder

Message 的拷贝操作:

```
class Message {  
public:  
    Message(const Message &other)  
        : m_contents{other.m_contents}, m_folders{other.m_folders} {  
        for (auto f : m_folders)  
            f->addMessage(*this);  
    }  
    ~Message() {  
        for (auto f : m_folders)  
            f->removeMessage(*this);  
    }  
};
```





Message and Folder

Message 的拷贝操作:

```
class Message {  
public:  
    Message &operator=(const Message &other) {  
        for (auto f : m_folders)  
            f->removeMessage(*this);  
        m_contents = other.m_contents;  
        m_folders = other.m_folders;  
        for (auto f : m_folders)  
            f->addMessage(*this);  
        return *this;  
    }  
};
```

避免重复: 写两个 `private` 函数来实现“将自身添加到所有文件夹/从所有文件夹中删除”的功能。



Message and Folder

Message 的**移动操作**：移动 `m_contents`，但我们仍然需要将自身从原来的文件夹中删除，以及添加到所有新的文件夹中。

很遗憾，**这个操作无法** `noexcept`。

```
class Message {  
public:  
    Message(Message &&other) : m_contents{std::move(other.m_contents)} {  
        moveFolders(other);  
    }  
private:  
    void moveFolders(Message &other) {  
        other.removeFromAll();  
        m_folders = std::move(other.m_folders);  
        addToAll();  
    }  
};
```



Message and Folder

Message 的**移动操作**：移动 `m_contents`，但我们仍然需要将自身从原来的文件夹中删除，以及添加到所有新的文件夹中。

很遗憾，**这个操作无法** `noexcept`。

```
class Message {  
public:  
    Message &operator=(Message &&other) {  
        if (this != &other) {  
            removeFromAll();  
            m_contents = std::move(other.m_contents);  
            moveFolders(other);  
        }  
        return *this;  
    }  
};
```




循环依赖

再次出现问题：现在我们不得不使用 `Folder` 的成员函数，但这要求 `Folder` 不是一个不完全类型（其定义必须已经给出）。

如果交换两个类的顺序，同样的问题会在 `Folder` 中发生。





循环依赖

出现了问题：现在我们不得不使用 `Folder` 的成员函数，但这要求 `Folder` 不是一个不完全类型（其定义必须已经给出）。

如果交换两个类的顺序，同样的问题会在 `Folder` 中发生。

解决方案：**在类内声明函数，等到 `Folder` 的定义给出之后再定义。**





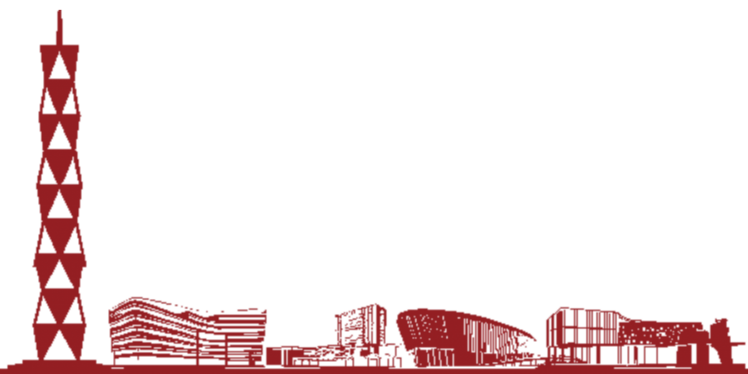
循环依赖

```
class Message {  
    private:  
        void addToAll();  
        void removeFromAll();  
};  
class Folder {  
    // ...  
};  
void Message::addToAll() {  
    for (auto f : m_folders)  
        f->addMessage(&this);  
}  
void Message::removeFromAll() {  
    for (auto f : m_folders)  
        f->removeMessage(&this);  
}
```



上海科技大学
ShanghaiTech University

多文件和动态链接



立志成才 报国裕民



#include

#include 机制：在其它真正的编译过程开始之前，由**预处理器**（preprocessor）将被#include 的文件原封不动地复制过来。

所以如果一个文件被 #include 两遍，它所包含的内容就会出现两次。

- 如果不加以保护，就会出现**重复定义符号**的问题。





Include guard

```
#ifndef MY_FILE_HPP
#define MY_FILE_HPP

// 将头文件的内容放在这里

#endif // MY_FILE_HPP
```

- `#ifndef X ... #endif` : 如果宏 `X` 在此前没有定义过，那么编译这部分代码。
- 如果 `MY_FILE_HPP` 此前没有定义过：首先在这里定义它，然后编译下面的代码。
- 当这段代码再次出现时，宏 `MY_FILE_HPP` 已经被定义过了！





分离式编译

例：将函数的声明写在头文件里，定义写在另一个 `.cpp` 文件里。

在 `a.cpp` 里调用函数 `sum` 时，只要 `sum` 的声明已经出现，其实就可以编译，只是暂时无法生成可执行文件。

- 因此只要在 `a.cpp` 里 `#include "sum.hpp"`。

编译： `g++ a.cpp sum.cpp -o a`

- 编译器会将 `a.cpp` 中对于 `sum` 的调用和在 `sum.cpp` 中的定义链接（Link）起来。





分离式编译

更进一步：可以单独编译 `sum.cpp`，生成一个中间文件，再在编译 `a.cpp` 时链接它：

```
g++ -shared sum.cpp -o libsum.so
```

这时生成了 `libsum.so`：**动态链接库文件**（有点儿像 Windows 上的 `.dll`，见过吗？）

编译 `a.cpp`：

```
g++ a.cpp -o a -Wl,-rpath . -L. -lsum
```

- `-Wl,-rpath .` 告知链接器在当前目录（`.`）下寻找 `.so` 文件
- `-L. -lsum` 链接到 `libsum.so`。

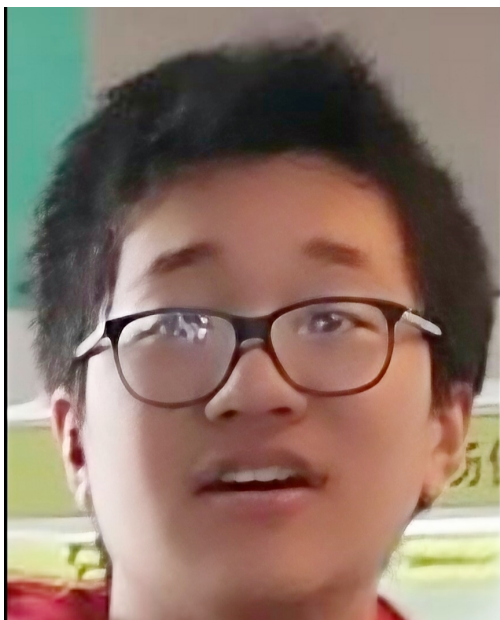




分离式编译

更进一步：可以单独编译 `sum.cpp`，生成一个中间文件，再在编译 `a.cpp` 时链接它。

好处？





分离式编译

更进一步：可以单独编译 `sum.cpp`，生成一个中间文件，再在编译 `a.cpp` 时链接它。

- 如果要修改 `sum.cpp` 中的具体实现，只需要重新编译 `sum.cpp`，不需要重新编译 `a.cpp`！（试一试）





分离式编译

C++ 项目的编译可能非常慢：在实际开发中，一次编译耗时半个小时以上是很常见的。

合理分离各个部分的代码可以有效提升代码的复用性、开发效率等等。

- 如果有 N 个程序都使用了 `sum`，只需让它们都链接到 `libsum.so`，而不是在编译时都带着 `sum.cpp` 一起编译。





分离式编译

将类的定义（包括其成员的声明）写在头文件里，将成员函数的具体实现写在 `.cpp` 里

- 循环依赖可以得到解决：在每个 `.cpp` 里 `#include` 所需要的头文件即可，可以保证使用一个类型时它已经定义完毕。
- 编译出来的动态库可以重复使用，可以被多个文件链接。

如果你看看 C 标准库的头文件，会发现几乎找不到函数定义，全是声明。

但是 C++ 标准库文件里却有很多函数给出了定义，这是为什么？





分离式编译

将类的定义（包括其成员的声明）写在头文件里，将成员函数的具体实现写在 `.cpp` 里

- 循环依赖可以得到解决：在每个 `.cpp` 里 `#include` 所需要的头文件即可，可以保证使用一个类型时它已经定义完毕。
- 编译出来的动态库可以重复使用，可以被多个文件链接。

如果你看看 C 标准库的头文件，会发现几乎找不到函数定义，全是声明。

但是 C++ 标准库文件里却有很多函数给出了定义，这是为什么？

- 在有 `template` 之前，一切都是很美好的...





Inline (内联) 函数

函数调用是有开销的!

能不能让编译器把 `a = std::min(b, c)` 变成 `a = b < c ? b : c` ?

- 将一些短的函数在调用点内联展开, 但又不像 `#define` 那样出现语法/语义的问题

```
inline int max(int a, int b) {  
    return a < b ? b : a;  
}
```





inline

`inline` 关键字：向编译器发出一个请求，希望将这个函数在调用点内联展开。

编译器可以拒绝某些 `inline` 请求，也可能自动为某些没有显式 `inline` 的函数 inline。

- 递归函数是难以 inline 的：编译器怎么知道它要展开多少层呢？
- 太过复杂的函数的 inline 请求也会被拒绝。

拒绝并不会导致报错或 warning。

任何写在类内的函数都是隐式 `inline` 的： `static` members, non-`static` members



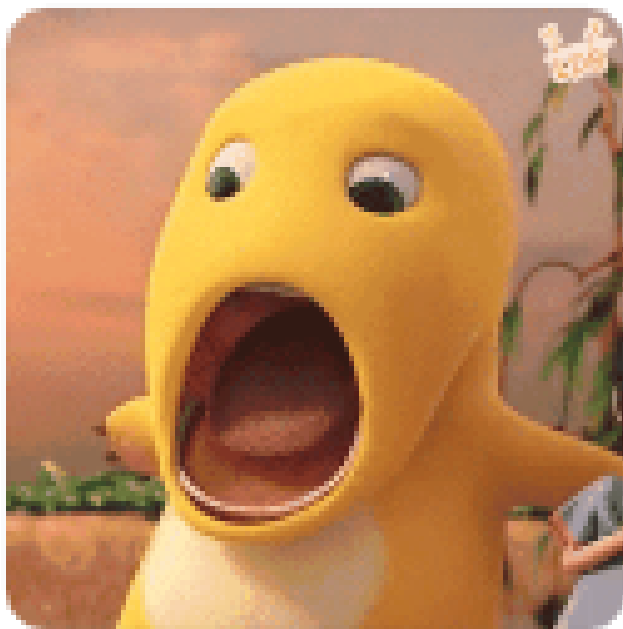


`inline`

直接把所有函数都加上 `inline` 不好吗？



【管理员】群出生，陪妹妹不陪兄弟





inline

直接把所有函数都加上 `inline` 不好吗？

首先，我们只需关注那些编译器**真的会** inline 的函数：

- 将一个函数 inline 会使得它被复制到调用点。如果这个函数在 200 个不同的地方被调用，这个函数的代码就被复制了 200 份——**代码膨胀**。
- 计算机执行指令时会把程序加载到内存里，因此代码膨胀就和你在内存里加载了过大的对象是一样的：Cache miss 增多，内存换页等问题（一年后的CS110：你们将会与这些问题正面作战！）。
- 对于需要预先编译然后动态链接的函数，如果编译器不为它生成函数本体，会导致链接错误。

《Effective C++》条款 30

