Lab3 Document

任务 1：理解内部连接和外部连接

任务 1.1：

运行结果如下：

```
D:\LAB\C++\ooplab3\cmake-build-debug\ooplab3.exe
0
2

Process finished with exit code 0
```

解释原因：

（1）打印的第一个是变量 variable 的值，此变量在头文件 header.h 中声明和定义，是一个静态的 int 类型的变量，在头文件和引入该头文件的文件中可以被访问到。

虽然 file1 和 file2 中的两个函数都对 variable 的值做了修改，但是通过打印他们的地址可以知道，在 file1.cpp、file2.cpp 和 main.cpp 中使用的 variable 的地址不同，也就是说，三个文件中使用的 variable 并不是同一块内存空间，对他们的访问互不影响。

```
int main() {
    function1();
    function2();

    std::cout << variable << std::endl;
    std::cout << "variable in main: " << &variable << std::endl;
    std::cout << variable2 << std::endl;
    return 0;

}
```

```
void function1() {
    variable = 1;
//    std::cout << variable << std::endl;
    std::cout << "variable in file1: " << &variable << std::endl;
    variable2 = 1;
}
```

```
void function2() {
    variable = 2;
//    std::cout << variable << std::endl;
    std::cout << "variable in file2: " << &variable << std::endl;
    variable2 = 2;

}
```

```
D:\LAB\C++\ooplab3\cmake-build-debug\ooplab3.exe
variable in file1: 0x40702c
variable in file2: 0x407034
0
variable in main: 0x407028
2

Process finished with exit code 0
```

因此，在 main.cpp 中打印的 variable 仍然是在 header.h 头文件中定义的初始值，为 0

（2）打印的第二个值是变量 variable2 的值，此变量也是在 header.h 头文件中定义，但是不同的是，该变量是 extern 的 int 类型的变量，它告诉编译器，存在这样一个变量 variable2，即使在当前编译的单元中没有看到它，它可能在另一个文件或者当前文件的后面定义。

在这个程序中，variable2 是在 main.cpp 的全局被定义，初始值为 0，之后在 file1.cpp 和 file2.cpp 中先后被修改为 1 和 2，打印出来的是最新的一次修改，也就是最终结果，2。

（1）和（2）有一个不同之处在于，variable 在每一个引用的文件中都有不同的内存地址，也就是在三个文件中都被定义，而 variable2 在三个文件中是一样的，都是在 main.cpp 中全局位置上声明的 variable2。

任务 1.2：

在 header.h 中改为“extern int variable2 = 2;”之后，对 main.cpp 的**编译不通过**，原因为：修改之后，header.h 头文件中已经定义了 variable2，而在 main.cpp 又通过 int variable2;再次定义了 variable2，这样的重复定义，在 C++中是不允许的

```
D:\LAB\C++\ooplab3>g++ -E main.cpp -o main.i

D:\LAB\C++\ooplab3>g++ -S main.i -o main.s
In file included from main.cpp:2:0:
header.h:6:12: warning: 'variable2' initialized and declared 'extern'
 extern int variable2 = 0;
            ^~~~~~~~~
main.cpp:5:5: error: redefinition of 'int variable2'
 int variable2;
     ^~~~~~~~~
In file included from main.cpp:2:0:
header.h:6:12: note: 'int variable2' previously defined here
 extern int variable2 = 0;
            ^~~~~~~~~
```

任务 1.3：

连接时出现错误：“variable2 未定义”，原因是：在 header.h 中，variable2 只有声明，没有定义，在 main.cpp 中将“int variable2”删掉之后，编译是可以

通过的，但是到连接时就会发现 variable2 在个编译单元中都没有被定义，出现错误

```
[ 25%] Building CXX object CMakeFiles/ooplab3.dir/main.cpp.obj
[ 50%] Building CXX object CMakeFiles/ooplab3.dir/file1.cpp.obj
[ 75%] Building CXX object CMakeFiles/ooplab3.dir/file2.cpp.obj
[100%] Linking CXX executable ooplab3.exe
CMakeFiles\ooplab3.dir/objects.a(main.cpp.obj): In function `main':
D:/LAB/C++/ooplab3/main.cpp:8: undefined reference to `variable2'
D:/LAB/C++/ooplab3/main.cpp:14: undefined reference to `variable2'
CMakeFiles\ooplab3.dir/objects.a(file1.cpp.obj): In function `Z9function1v':
D:/LAB/C++/ooplab3/file1.cpp:9: undefined reference to `variable2'
CMakeFiles\ooplab3.dir/objects.a(file2.cpp.obj): In function `Z9function2v':
D:/LAB/C++/ooplab3/file2.cpp:9: undefined reference to `variable2'
collect2.exe: error: ld returned 1 exit status
mingw32-make.exe[3]: *** [ooplab3.exe] Error 1
```

任务 2：理解名字空间

任务 2.1：

两个头文件：header1.h 和 header2.h，header1.h 文件中声明了名字空间 header1，header2.h 中声明了名字空间 header2：

```
#ifndef OOPLAB3_HEADER1_H
#define OOPLAB3_HEADER1_H

namespace header1{
    int x = 1;
    int y = 1;
}

#endif //OOPLAB3_HEADER1_H
```

```
#ifndef OOPLAB3_HEADER2_H
#define OOPLAB3_HEADER2_H

namespace header2{
    int x = 2;
    float y = 2.0;
}

#endif //OOPLAB3_HEADER2_H
```

main.cpp，将两个名字空间的 x 和 y 分别打印出来：

```
#include <iostream>
#include "header1.h"
#include "header2.h"

int main() {
    std::cout << "x in header1.h: " << header1::x << std::endl;
    std::cout << "x in header2.h: " << header2::x << std::endl;

    std::cout << "y in header1.h: " << header1::y << std::endl;
    std::cout << "y in header2.h: " << header2::y << std::endl;
}
```

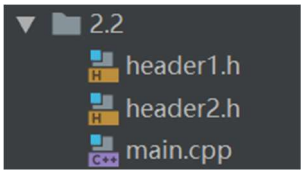
运行结果如下：

```
D:\LAB\C++\ooplab3\cmake-build-debug\ooplab3.exe
x in header1.h: 1
x in header2.h: 2
y in header1.h: 1
y in header2.h: 2

Process finished with exit code 0
```

说明无论变量的类型一样或者不一样，不同名字空间中相同的标识符并不会导致重复定义或声明

任务 2.2：



两个头文件 header1.h 和 header2.h，都声明了相同的名字空间：header：

```h
#ifndef OOPLAB3_HEADER1_H
#define OOPLAB3_HEADER1_H

namespace header{
    int a = 1;
    int b = 2;
}

#endif //OOPLAB3_HEADER1_H
```

```h
#ifndef OOPLAB3_HEADER2_H
#define OOPLAB3_HEADER2_H

namespace header{
    int b = 3;
    int c = 4;
}

#endif //OOPLAB3_HEADER2_H
```

main.cpp 将 a b c 打印出来：

```cpp
#include <iostream>
#include "header1.h"
#include "header2.h"

#define F(x) std::cout << #x " = " << x << std::endl

int main() {
    using namespace header;
    F(a);
    F(b);
    F(c);
}
```

尝试运行一下，报错，变量 b 重复定义：

```
Scanning dependencies of target ooplab3
[ 50%] Building CXX object CMakeFiles/ooplab3.dir/2.2/main.cpp.obj
In file included from D:\LAB\C++\ooplab3\2.2\main.cpp:7:0:
D:\LAB\C++\ooplab3\2.2\header2.h:9:9: error: redefinition of 'int header::b'
    int b = 3;
        ^
In file included from D:\LAB\C++\ooplab3\2.2\main.cpp:6:0:
D:\LAB\C++\ooplab3\2.2\header1.h:10:9: note: 'int header::b' previously defined here
    int b = 2;
        ^
```

这是因为，虽然 header 在两个文件中都有声明，但是这两个声明是会存在 merge 的，那么两处对同一个变量声明和定义两次，编译器就会报错。
倘若把 header2 头文件中的“int b = 3;”改为“b = 3;”，依然会报错：

```
Scanning dependencies of target ooplab3
[ 50%] Building CXX object CMakeFiles/ooplab3.dir/2.2/main.cpp.obj
In file included from D:\LAB\C++\ooplab3\2.2\main.cpp:7:0:
D:\LAB\C++\ooplab3\2.2\header2.h:9:5: error: 'b' does not name a type
     b = 3;
     ^
```

原因是 C++的名字空间中，每一个标识符都必须有类型

任务 3：对比 Typescript

1.名字空间：

1.1 创建一个名字空间

（1）C++中创建一个名字空间：

```
namespace MyLib {
  // Declarations
}
```

Typescript 中创建一个名字空间：

```
namespace SomeNameSpaceName {
    // Declarations
}
```

（2）C++和 Typescript 中都支持两个名字相同的名字空间合并：C++的代码可以参考任务 2.2，typescript 的示例代码：

```
namespace namespace_name1 {
    export namespace namespace_name2 {
        export class class_name {}
    }
}
```

（3）C++中一个名字空间可以使用另一个名字做它的别名，这在 typescript 中也是允许的：

```
namespace BobsSuperDuperLibrary {
  class Widget { /* ... */ };
  class Poppit { /* ... */ };
  // ...
}
// Too much to type! I'll alias it:
namespace Bob = BobsSuperDuperLibrary;
```

```
namespace Drawing {

}
import x = Drawing
```

（ts）

1.1.1 未命名的名字空间

C++的每个翻译单元都可以包含一个未命名的名字空间，每个翻译单元中未命名的名字空间最多只有一个，但是这在 typescript 中是不允许的：

```
namespace {
  class Arm  { /* ... */ };
  class Leg  { /* ... */ };
  class Head { /* ... */ };
  class Robot {
    Arm arm[4];
    Leg leg[16];
    Head head[3];
    // ...
  } xanthan;
  int i, j, k;
}
```

1.1.2 友元

C++中可以在名字空间的类定义之内插入一个友元声明，而在 typescript 中没有看到类似友元的概念：

```
namespace Me {
  class Us {
    //...
    friend void you();
  };
}
```

1.2 使用名字空间

1.2.1 C++中使用名字空间

C++在一个名字空间中引用一个名字支持两种方法：第一种是作用域运算符，第二种是 using 指令：

1.2.1.1 作用域运算符

类似与 C++中的类：

```
namespace X {
  class Y {
    static int i;
  public:
    void f();
  };
}
```

1.2.1.2 使用指令

可以在头文件(最好不要)、cpp 文件、函数内部和其他名字空间中使用 using 指令引用一个名字空间中的所有名字：

```
using namespace std;
```

1.2.1.3 使用声明

有时候并不想引用一个名字空间中的所有名字，就可以使用类似与 using std::cout 的语法来引入一个名字，使用该语法还可以将不同名字空间中的名字结合成为一个新的名字空间：

```
namespace Q {
  using U::f;
  using V::g;
  // ...
}
```

该例子中就是名字空间 U 中的 f 和 V 中的 g 合并在一起形成名字空间 Q

1.2.2 typescript 中使用名字空间：

typescript 中通过类似/// <reference path = "IShape.ts" />的语句引入 IShape.ts 文件中的名字空间：

```
//IShape.ts
namespace Drawing {
    export interface IShape {
        draw();
    }
}
```

```
//name.ts
/// <reference path = "IShape.ts" />
namespace Drawing {
    export class Circle implements IShape {
        public draw() {
            console.log("Circle is drawn");
        }
    }
}
```

2. Typescript 模块和 C++连接：

2.1 typescript 中的模块：

Typescript 中模块是在其自身的作用域里执行，并不是在全局作用域，这意味着定义在模块里面的变量、函数和类等在模块外部是不可见的，除非明确地使用 export 导出它们。类似地，我们必须通过 import 导入其他模块导出的变量、函数、类等：

```
// 文件名 : SomeInterface.ts
export interface SomeInterface {
    // 代码部分
}
```

```
import someInterfaceRef = require("./SomeInterface");
```

（请忽略报错）

2.2 C++中的内部连接和外部连接

C++与模块相对应的就是编译单元，假如一个名称对于他的编译单元来说是局部的，并且在连接时不会和其他编译单元中的同样的名称相冲突，那么这个名称有内部连接；假如一个名称在连接时能够和其他编译单元交互，那么这个名称就有外部连接。

内部连接：在文件作用域内，一个被明确声明为 static 的对象或函数的名字对编译单元来说是局部于编译单元的，这些名字有内部连接，像 inline 等也是内部连接

外部连接：一般情况下，在文件作用域内的所有名字（不嵌套在类或函数中的名字）在连接时对所有的编译单元来说都是可见的，这些就是外部连接

C++的外部连接是通过 include 实现的