南京大学

计算机科学与技术系

软件工程实验报告

实验名称：软件测试与修复

学　　号：

姓　　名：

指导教师：王豫

实验地点：

实验时间： 2023.1.2

一、开源项目选择

由于实验 4 选择的开源项目体量过大，理解源码，进行测试与修复相对困难，我重新选择了以下两个小型开源项目，和自己的项目作为实验对象。
项目 1：json-parser 一个由 c++编写的，源码两千行左右的 JSON 文档解析器。Json-parser 可以解析 JSON 文档产生 JSON value 并返回 JSON value 对象。
项目地址：https://github.com/Barenboim/json-parser/tree/main

项目 2：PicoHTTPParser 一个小型的 HTTP 解析器，适用于 C 语言。代码行数 500 行左右，star 数 1.6k。项目地址：https://github.com/h2o/picohttpparser

项目 3：在实验 1-4 中完成的医院信息管理 c++项目。

二、单元测试报告

1. 测试目的

单元测试是软件开发中的一种重要测试方法，其目的是对软件中的最小可测试单元进行检查和验证。通过单元测试，可以确保每个单元（如函数、方法）按照预期工作，从而提高代码的质量和可靠性。单元测试还有助于及早发现和解决问题，减少软件在后期开发中的风险和成本。

2. 测试环境

在本次实验中，由于项目编程语言选择的是 c++，我主要使用 Google Test 作为测试工具。Google Test（GTEST）是一个开源的单元测试框架，由 Google 公司开发。它适用于编写和运行 C++代码的单元测试，具有简洁易用、功能强大等特点。GTEST 提供了丰富的断言库，可以方便地检查各种条件；同时，它还支持自定义断言，以满足特定的测试需求。此外，GTEST 还提供了测试发现、测试报告等特性，使得测试过程更加高效和直观。

3. 测试工具安装

我直接利用 visual studio 中包含的 gtest 测试项目，建立测试用例进行测试。编写的测试用例示例如下：

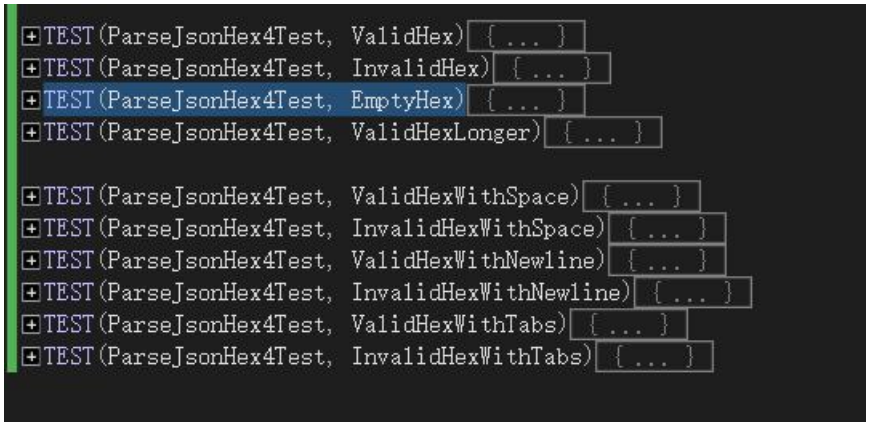
```
TEST(ParseJsonUnicode, Unicode01) {
    const char* input = "0041";
    char utf8[4];
    EXPECT_EQ(1, __parse_json_unicode(input, &input, utf8));
    EXPECT_EQ("A"[0], utf8[0]);
}
```

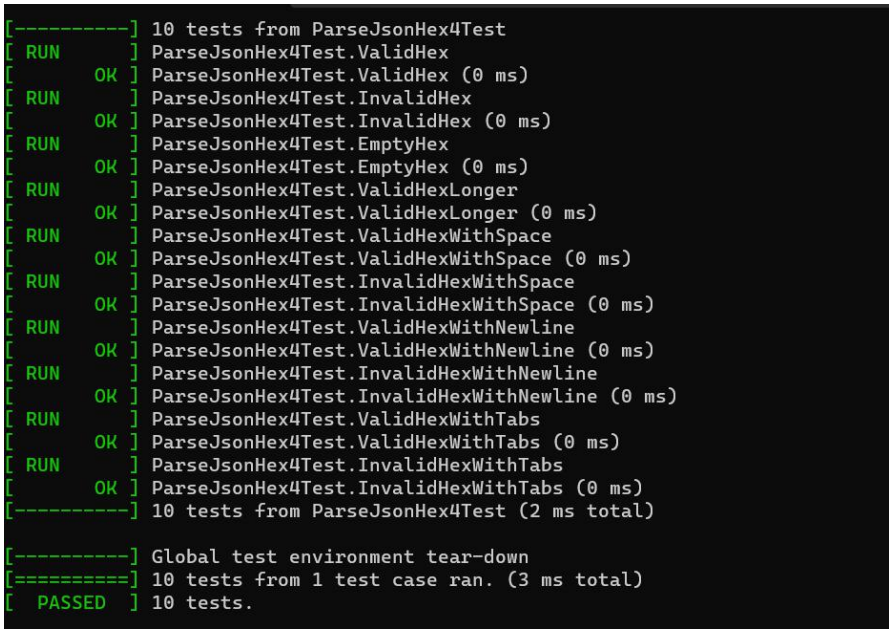
4. 测试报告

（1）Json-parser 测试

功能 1：_parse_json_hex4：此函数用于解析 JSON 字符串中的前 4 位十六进制表示的数字。此函数的目的是将这样的十六进制字符串转换为对应的十进制数值。
针对这个函数，我编写了 10 个 Gtest 测试用例，考虑不同的输入情况，包括有效的十六进制字符串、无效的十六进制字符串（如包含非法字符），过长的字符串以及空字

符串等等，完善考虑了有效等价类和无效等价类的情况。
其中，第 1 条测试用例设置了有效字符串的情况，第 2 条设置成包含了无效字符的情况，第 3-4 条，设置成不符合解析要求的十六进制字符串，第 5-10 条设置成带有分隔符的十六字符串情况。除第一条用例之外，其他用例的结果应该为无法解析。
编写源码和测试结果如下：





功能 2：__parse_json_unicode：此函数用于解析 JSON 中的十六进制表示的 Unicode 字符,将其转换为 utf-8 字符。它接收一个指向 JSON 字符串的指针，一个指向字符串末尾的指针，以及一个用于存储解析结果的 Unicode 码点。函数首先将接收的十六进制数字转换为 Unicode 码点，然后返回解析结果。如果在解析过程中遇到非法字符，函数将返回错误码。
针对这个函数，编写 GTest 测试用例 10 个，我考虑了不同的输入情况，包括合法的 UTF-8 编码序列、非法的编码序列、以及边界条件等待，其中，测试用例 1,2,3,4 用实际的 Unicode 编码来填充期望值。同时，测试用例 6 和 7 构造非法的 UTF-8 序列。测试用例 9 和 10 是边界条件测试，它们验证函数对于空字符串和极大字符串的处理。
编写源码和测试结果如下：
经分析，造成测试未能通过的主要原因是源码没有对无法转换成 UTF-8 的编码做特殊处理，导致与结果不符。

```
  //测试用例 1：Unicode字符（1字节）
⊞TEST(ParseJsonUnicode, Unicode01) { ... }
  //测试用例 1：Unicode字符（3字节）

⊞TEST(ParseJsonUnicode, Unicode03) { ... }
  // 测试用例 3：正常的UTF-8字符（2字节）
⊞TEST(ParseJsonUnicode, Unicode02) { ... }
  // 测试用例 4：正常的UTF-8字符（4字节）
⊞TEST(ParseJsonUnicode, Unicode04) { ... }
  // 测试用例 5：正常的UTF-8字符（4字节）
⊞TEST(ParseJsonUnicode, Utf8FourBytes) { ... }
  // 测试用例 6：非法的UTF-8序列（非法的字节开头）
⊞TEST(ParseJsonUnicode, IllegalUtf8Sequence) { ... }
  // 测试用例 7：非法的UTF-8序列（无效的编码）
⊞TEST(ParseJsonUnicode, InvalidUtf8Encoding) { ... }
  // 测试用例 8：非法的UTF-8序列（超出范围）
⊞TEST(ParseJsonUnicode, OutOfRangeUtf8) { ... }
  // 测试用例 9：边界条件：空字符串
⊞TEST(ParseJsonUnicode, EmptyString) { ... }
  // 测试用例 10：边界条件：非常大的字符串
⊞TEST(ParseJsonUnicode, LargeString) { ... }
```

```
  __parse_json_unicode(input, &input, utf8)
    Which is: -2
[  FAILED  ] ParseJsonUnicode.Utf8FourBytes (1 ms)
[ RUN      ] ParseJsonUnicode.IllegalUtf8Sequence
[       OK ] ParseJsonUnicode.IllegalUtf8Sequence (0 ms)
[ RUN      ] ParseJsonUnicode.InvalidUtf8Encoding
[       OK ] ParseJsonUnicode.InvalidUtf8Encoding (0 ms)
[ RUN      ] ParseJsonUnicode.OutOfRangeUtf8
[       OK ] ParseJsonUnicode.OutOfRangeUtf8 (0 ms)
[ RUN      ] ParseJsonUnicode.EmptyString
D:\cpp\Sample-Test1\Sample-Test1\test.cpp(281): error: Expected equality of these values:
  0
  __parse_json_unicode(input, &input, utf8)
    Which is: -2
[  FAILED  ] ParseJsonUnicode.EmptyString (0 ms)
[ RUN      ] ParseJsonUnicode.LargeString
[       OK ] ParseJsonUnicode.LargeString (0 ms)
[----------] 10 tests from ParseJsonUnicode (6 ms total)

[----------] Global test environment tear-down
[==========] 10 tests from 1 test case ran. (7 ms total)
[  PASSED  ] 7 tests.
[  FAILED  ] 3 tests, listed below:
[  FAILED  ] ParseJsonUnicode.Unicode04
[  FAILED  ] ParseJsonUnicode.Utf8FourBytes
[  FAILED  ] ParseJsonUnicode.EmptyString

 3 FAILED TESTS
```

（2）PicoHTTPParser：

功能 1：首先针对项目提供的 phr_parse_request 函数，即解析 http request 的函数进行单元测试。10 个测试用例分别检查了缓冲区为空，请求有效，请求无效，版本号是否正确解析等等。

10 个测试用例中，测试用例示例如图：

```
TEST(ParseRequestTest, TestEmptyBuffer) {
    const char* buf = "";
    const char* buf_end = buf;
    const char** method = NULL;
    size_t* method_len = NULL;
    const char** path = NULL;
    size_t* path_len = NULL;
    int* minor_version = NULL;
    struct phr_header* headers = NULL;
    size_t* num_headers = NULL;
    size_t max_headers = 0;
    int* ret = NULL;
    EXPECT_EQ(false, parse_request(buf, buf_end, method, method_len, path, path_len, minor_version, headers, num_headers, max_headers, ret));
    EXPECT_EQ(*ret, -1);
}
```

```
TEST(ParseRequestTest, TestEmptyBuffer) { ... }
TEST(ParseRequestTest, TestInvalidMethod) { ... }
TEST(ParseRequestTest, TestInvalidPath) { ... }
TEST(ParseRequestTest, TestValidRequest) { ... }
TEST(ParseRequestTest, TestCRLEmptyLine) { ... }
TEST(ParseRequestTest, TestCRLFLine) { ... }
TEST(ParseRequestTest, TestMinorVersion) { ... }
TEST(ParseRequestTest, TestInvalidHTTPVersion) { ... }
TEST(ParseRequestTest, TestMajorVersion) { ... }
TEST(ParseRequestTest, TestHeaderParsing) { ... }
```

```
[----------] 10 tests from ParseRequestTest
[ RUN      ] ParseRequestTest.TestEmptyBuffer
[       OK ] ParseRequestTest.TestEmptyBuffer (0 ms)
[ RUN      ] ParseRequestTest.TestInvalidMethod
[       OK ] ParseRequestTest.TestInvalidMethod (0 ms)
[ RUN      ] ParseRequestTest.TestInvalidPath
[       OK ] ParseRequestTest.TestInvalidPath (0 ms)
[ RUN      ] ParseRequestTest.TestValidRequest
[       OK ] ParseRequestTest.TestValidRequest (0 ms)
[ RUN      ] ParseRequestTest.TestCRLEmptyLine
[       OK ] ParseRequestTest.TestCRLEmptyLine (0 ms)
[ RUN      ] ParseRequestTest.TestCRLFLine
[       OK ] ParseRequestTest.TestCRLFLine (0 ms)
[ RUN      ] ParseRequestTest.TestMinorVersion
[       OK ] ParseRequestTest.TestMinorVersion (0 ms)
[ RUN      ] ParseRequestTest.TestInvalidHTTPVersion
[       OK ] ParseRequestTest.TestInvalidHTTPVersion (0 ms)
[ RUN      ] ParseRequestTest.TestMajorVersion
[       OK ] ParseRequestTest.TestMajorVersion (0 ms)
[ RUN      ] ParseRequestTest.TestHeaderParsing
[       OK ] ParseRequestTest.TestHeaderParsing (0 ms)
[----------] 10 tests from ParseRequestTest (1 ms total)

[----------] Global test environment tear-down
[==========] 10 tests from 1 test case ran. (3 ms total)
[  PASSED  ] 10 tests.
```

功能 2：再针对项目提供的 parse_token 接口，进行单元测试。

```
TEST(ParseRequestTest, TestEmptyBuffer) { ... }
TEST(ParseRequestTest, TestSingleCharToken) { ... }
TEST(ParseRequestTest, TestMultiCharToken) { ... }
TEST(ParseRequestTest, TestTokenWithSpace) { ... }
TEST(ParseRequestTest, TestTokenWithSpecialChars) { ... }
TEST(ParseRequestTest, TestTokenAtBufferEnd) { ... }
TEST(ParseRequestTest, TestTokenAfterEmptyLine) { ... }
TEST(ParseRequestTest, TestTokenAfterNewLine) { ... }
TEST(ParseRequestTest, TestTokenWithNextChar) { ... }
TEST(ParseRequestTest, TestInvalidToken) { ... }
```

```
[----------] 10 tests from ParseRequestTest
[ RUN      ] ParseRequestTest.TestEmptyBuffer
[       OK ] ParseRequestTest.TestEmptyBuffer (0 ms)
[ RUN      ] ParseRequestTest.TestSingleCharToken
[       OK ] ParseRequestTest.TestSingleCharToken (0 ms)
[ RUN      ] ParseRequestTest.TestMultiCharToken
[       OK ] ParseRequestTest.TestMultiCharToken (0 ms)
[ RUN      ] ParseRequestTest.TestTokenWithSpace
[       OK ] ParseRequestTest.TestTokenWithSpace (0 ms)
[ RUN      ] ParseRequestTest.TestTokenWithSpecialChars
[       OK ] ParseRequestTest.TestTokenWithSpecialChars (0 ms)
[ RUN      ] ParseRequestTest.TestTokenAtBufferEnd
[       OK ] ParseRequestTest.TestTokenAtBufferEnd (0 ms)
[ RUN      ] ParseRequestTest.TestTokenAfterEmptyLine
[       OK ] ParseRequestTest.TestTokenAfterEmptyLine (0 ms)
[ RUN      ] ParseRequestTest.TestTokenAfterNewLine
[       OK ] ParseRequestTest.TestTokenAfterNewLine (0 ms)
[ RUN      ] ParseRequestTest.TestTokenWithNextChar
[       OK ] ParseRequestTest.TestTokenWithNextChar (0 ms)
[ RUN      ] ParseRequestTest.TestInvalidToken
[       OK ] ParseRequestTest.TestInvalidToken (0 ms)
[----------] 10 tests from ParseRequestTest (1 ms total)

[----------] Global test environment tear-down
[==========] 10 tests from 1 test case ran. (3 ms total)
[  PASSED  ] 10 tests.
```

（3）自身项目：

功能 1：针对编写的创建医院条目和打印医院信息进行单元测试。测试的目的主要是观察类构造函数是否能正确载入用户输入的信息。我编写了 5 条测试用例，由于函数没有分支，代码覆盖率达到 100%。测试用例和结果如下：

```
⊞TEST(HospitalTest, ConstructorAndMemberAccess) { ... }
⊞TEST(PrintHospitalDataTest, PrintHospitalData) { ... }
⊞TEST(PatientTest, ConstructorAndMemberAccess) { ... }
⊟TEST(PrintHospitalDataTest, PrintPatientData) {
    vector<Hospital> hospitals;
    hospitals.push_back(Patient("Patient1", 12345, "TestHospital1", "TestLocation", 50, 3.5f, "TestContact", "TestDoctor", 500));
    hospitals.push_back(Patient("Patient2", 12346, "TestHospital12", "TestLocation2", 75, 4.0f, "TestContact2", "TestDoctor2", 700));

    PrintHospitalData(hospitals);
}
⊟TEST(PrintHospitalDataTest, PrintEmptyHospitalData) {
    vector<Hospital> hospitals;
    PrintHospitalData(hospitals);
}
```

```
[ RUN      ] PrintHospitalDataTest.PrintPatientData
PRINT hospitals DATA:
HospitalName     Location     Beds_Available     Rating     Hospital_Contact     Doctor_Name     Price_Per_Bed
TestHospital                        TestLocation           50                    3.5            TestContact
 TestDoctor                         500
TestHospital2                        TestLocation2          75                    4             TestContact2
  TestDoctor2                        700


[       OK ] PrintHospitalDataTest.PrintPatientData (0 ms)
[ RUN      ] PrintHospitalDataTest.PrintEmptyHospitalData
PRINT hospitals DATA:
HospitalName     Location     Beds_Available     Rating     Hospital_Contact     Doctor_Name     Price_Per_Bed


[       OK ] PrintHospitalDataTest.PrintEmptyHospitalData (1 ms)
[----------] 3 tests from PrintHospitalDataTest (2 ms total)

[----------] 1 test from PatientTest
[ RUN      ] PatientTest.ConstructorAndMemberAccess
[       OK ] PatientTest.ConstructorAndMemberAccess (0 ms)
[----------] 1 test from PatientTest (0 ms total)

[----------] Global test environment tear-down
[==========] 5 tests from 3 test cases ran. (3 ms total)
[  PASSED  ] 5 tests.
```

可以看到，类正确打印了结果信息，对于空 vector，也没有发生崩溃。

功能 2：针对按床位价格排序医院的子功能进行单元测试，测试的目的主要是函数能否正确按照床位对医院信息进行排序，和函数能否应对异常情况。我针对几种情况，编写了 5 个 gtest 测试用例，代码覆盖率达到 100%。
测试用例和测试结果如下：

```
TEST(HospitalTest, Constructor) { ... }
TEST(HospitalTest, BedsPrice) { ... }
TEST(HospitalTest, SortByBedsPrice) {
    vector<Hospital> hospitals;
    hospitals.push_back(Hospital("Hospital1", "Location1", 100, 4.5, "Contact1", "Doctor1", 500));
    hospitals.push_back(Hospital("Hospital2", "Location2", 150, 4.0, "Contact2", "Doctor2", 600));
    SortByBedsPrice(hospitals);
    EXPECT_TRUE(beds_price(hospitals[0], hospitals[1]));
}
TEST(HospitalTest, SortByBedsPriceResult) {
    vector<Hospital> hospitals;
    hospitals.push_back(Hospital("Hospital1", "Location1", 100, 4.5, "Contact1", "Doctor1", 500));
    hospitals.push_back(Hospital("Hospital2", "Location2", 150, 4.0, "Contact2", "Doctor2", 600));
    SortByBedsPrice(hospitals);
    EXPECT_EQ(hospitals[0].H_name, "Hospital1");
    EXPECT_EQ(hospitals[1].H_name, "Hospital2");
}

TEST(HospitalTest, SortByBedsPriceDifferentData) {
    vector<Hospital> hospitals;
    hospitals.push_back(Hospital("Hospital1", "Location1", 100, 4.5, "Contact1", "Doctor1", 300));
    hospitals.push_back(Hospital("Hospital2", "Location2", 150, 4.0, "Contact2", "Doctor2", 200));
    hospitals.push_back(Hospital("Hospital3", "Location3", 50, 5.0, "Contact3", "Doctor3", 400));
    SortByBedsPrice(hospitals);
    EXPECT_EQ(hospitals[0].H_name, "Hospital1");
    EXPECT_EQ(hospitals[1].H_name, "Hospital2");
    EXPECT_EQ(hospitals[2].H_name, "Hospital3");
}
```

```
[       OK ] HospitalTest.SortByBedsPriceDifferentData (1 ms)
[----------] 5 tests from HospitalTest (5 ms total)

[----------] Global test environment tear-down
[==========] 5 tests from 1 test case ran. (6 ms total)
[  PASSED  ] 5 tests.
```

根据测试结果，可以看到排序函数并没有改变项目的真正顺序，这达到了功能与数据

的解耦合。
所有单元测试的整体覆盖率表格如下：

语句覆盖率	测试 1	测试 2
项目 1	100%	90%（有一个分支未覆盖）
项目 2	97.5%	91.42%
自身项目	100%	100%

三、集成测试报告

1. 测试目的

集成测试的主要目的是验证各个软件模块在组合后的整体功能是否满足预期的需求。通过集成测试，我们可以确保不同模块之间的接口交互正确无误，各个模块能够协同工作，从而保证整个软件系统的稳定性和可靠性。此外，集成测试还能够发现模块间集成过程中可能引入的新问题，如数据传递错误、资源竞争、死锁等。因此，集成测试是软件开发过程中至关重要的一环，它有助于提高软件质量，减少系统级缺陷，降低后期维护成本。

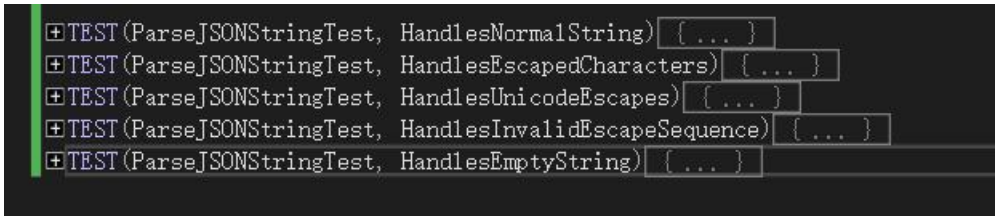
2. 测试方法

在实验中，我仍然使用 gtest 框架，对单元测试的功能进行集成测试，我将多个功能进行组合，使用 gtest 断言观察其结果，以达到测试目的。

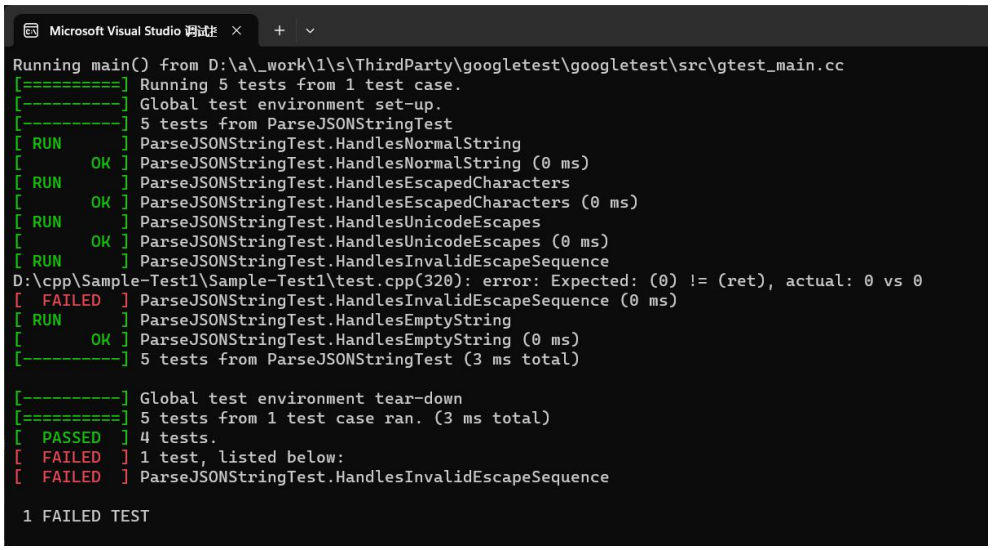
3. 测试报告

（1）项目 1：

第一组：自底向上测试：针对单元测试的两个子功能，进行两个功能的集成测试。我利用项目中集成两个功能的__parse_json_string 函数，这个函数集成了单元测试中两个子功能，可以对 JSON 文件里的字符串进行总体解析。
我针对这个函数，编写了 5 条测试用例，其中有 1 条测试了非法字符串的情况：



集成测试结果：

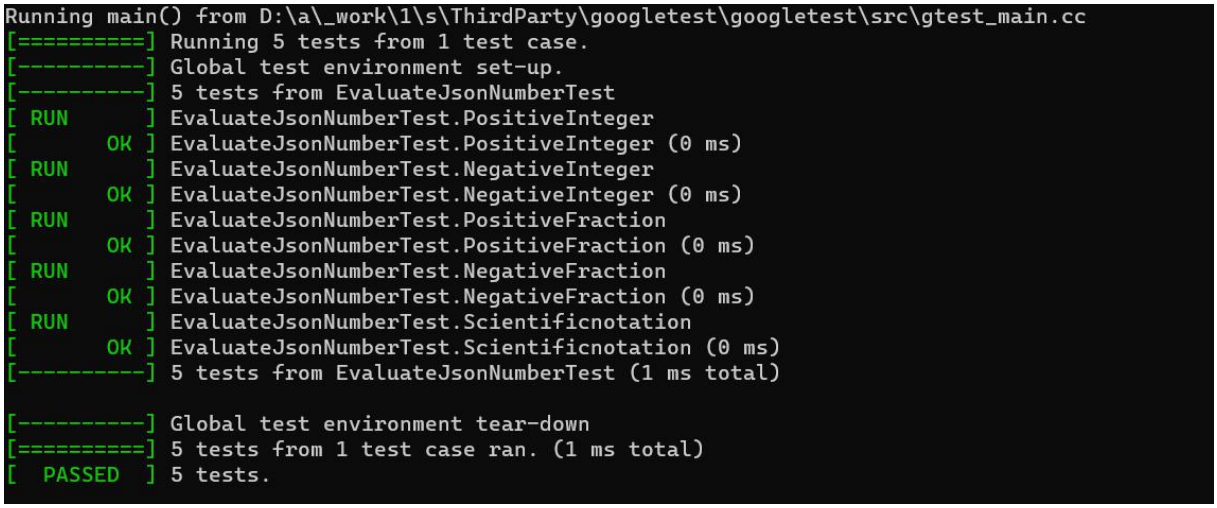


单元测试中发现的，无法处理非法字符串的缺陷，在集成测试中仍然显现。

第二组：整体功能测试。针对项目中另外两个子功能的集成功能：解析 JSON 文件中数字的上层功能，编写 5 个 Gtest 测试用例，并进行集成测试。这些测试用例涵盖了函数的几种常见使用场景，包括正整数、负整数、正分数、负分数和科学计数法。

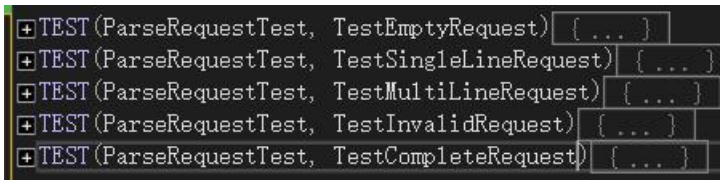
```
TEST(EvaluateJsonNumberTest, PositiveInteger) {
    EXPECT_DOUBLE_EQ(__evaluate_json_number("123", "", 0), 123.0);
}
TEST(EvaluateJsonNumberTest, NegativeInteger) {
    EXPECT_DOUBLE_EQ(__evaluate_json_number("-123", "", 0), -123.0);
}
TEST(EvaluateJsonNumberTest, PositiveFraction) {
    EXPECT_DOUBLE_EQ(__evaluate_json_number("0", "123", 0), 0.123);
}
TEST(EvaluateJsonNumberTest, NegativeFraction) {
    EXPECT_DOUBLE_EQ(__evaluate_json_number("0", "-123", 0), 0);
}
TEST(EvaluateJsonNumberTest, Scientificnotation) {
    EXPECT_DOUBLE_EQ(__evaluate_json_number("1", "234", 6), 1.234e6);
}
```

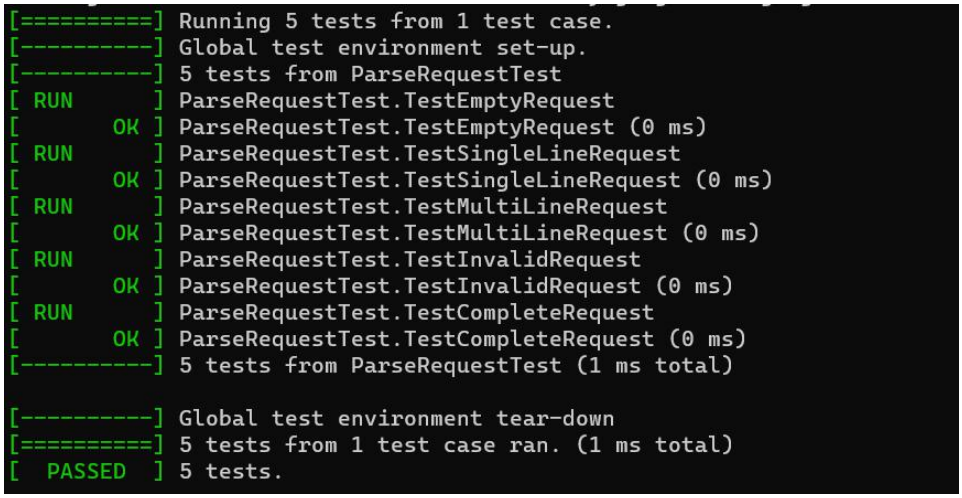
测试结果如下图，测试用例均通过。



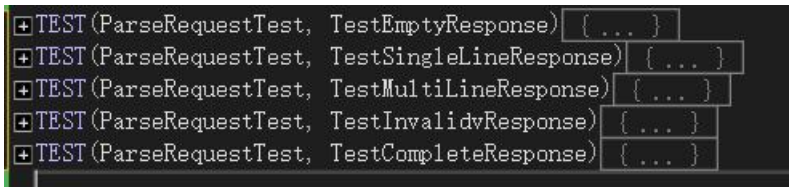
项目 2：

第一组：针对单元测试中，测试的两个子功能 parse_http_request 和 parse_http_token，进行功能的集成测试，编写 gtest 测试用例，设计思路如下：
TestEmptyRequest: 测试当输入缓冲区为空时，http_parser 函数的行为是否正确。预期函数应该返回一个错误码，因为无法从空缓冲区解析 HTTP 请求。
TestSingleLineRequest: 测试当输入缓冲区中只有一个请求行时，http_parser 函数的行为是否正确。预期函数应该能够正确解析这个单行请求。
TestMultiLineRequest: 测试当输入缓冲区中包含一个多行请求时，http_parser 函数的行为是否正确。预期函数应该能够正确解析这个多行请求。
TestInvalidRequest: 测试当输入缓冲区中包含一个无效的 HTTP 请求时，http_parser 函数的行为是否正确。预期函数应该返回一个错误码，因为请求不符合预期的格式。
TestCompleteRequest: 测试当输入缓冲区中包含一个完整的 HTTP 请求时，http_parser 函数的行为是否正确。预期函数应该能够正确解析这个完整的请求，并填充相应的解析结果结构。





第二组：针对项目中包含的 parse_http_response 功能，测试项目对 http 响应的解析效果。进行另一组集成测试，我用 gtest 编写了 5 个测试用例，设计思路与第一组测试 http request 相似。



```
[==========] Running 5 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 5 tests from ParseRequestTest
[ RUN      ] ParseRequestTest.TestEmptyResponse
[       OK ] ParseRequestTest.TestEmptyResponse (0 ms)
[ RUN      ] ParseRequestTest.TestSingleLineResponse
[       OK ] ParseRequestTest.TestSingleLineResponse (0 ms)
[ RUN      ] ParseRequestTest.TestMultiLineResponse
[       OK ] ParseRequestTest.TestMultiLineResponse (0 ms)
[ RUN      ] ParseRequestTest.TestInvalidvResponse
[       OK ] ParseRequestTest.TestInvalidvResponse (0 ms)
[ RUN      ] ParseRequestTest.TestCompleteResponse
[       OK ] ParseRequestTest.TestCompleteResponse (0 ms)
[----------] 5 tests from ParseRequestTest (1 ms total)

[----------] Global test environment tear-down
[==========] 5 tests from 1 test case ran. (2 ms total)
[  PASSED  ] 5 tests.
```

自身项目：

第一组：针对单元测试的两个子功能，进行集成测试，在测试中，先后执行测试的两个函数，并观察结果。我用 gtest 编写了个集成测试用例，其中，3 个测试了医院类的构造函数，打印功能和排序功能，2 个用例测试了病人类的构造函数和排序功能。

```
⊞TEST(HospitalTest, ConstructAndPrintData) { ... }

⊞TEST(PatientTest, ConstructAndPrintData) { ... }

⊞TEST(HospitalTest, SortByBedsPrice) { ... }

⊞TEST(PatientTest, SortByBedsPrice) { ... }

⊞TEST(HospitalTest, PrintDataOutput) { ... }
```

5 个测试均通过：

```
[       OK ] PatientTest.ConstructAndPrintData (1 ms)
[ RUN      ] PatientTest.SortByBedsPrice
SORT BY Available Beds Price:

PRINT hospitals DATA:
HospitalName     Location     Beds_Available     Rating     Hospital_Contact     Doctor_Name     Price_Per_Bed
医院A                         位置A          100                    4.5           联系方式A              医生A
                1000
医院C                         位置C          150                    4.7           联系方式C              医生C
                1200
医院B                         位置B          200                    4           联系方式B              医生B
               1500
```

第二组：针对我编写项目的其他功能，如按床铺排序医院，按城市排序医院等功能进行集成测试，保证每个测试用例包含两个以上的功能，共设计 5 个测试用例如下：

```
⊞TEST(HospitalTest, PrintAndSortByBedsPrice) { ... }
⊞TEST(HospitalTest, SortByNameAndPrint) { ... }
⊞TEST(HospitalTest, SortByBedsAvailableAndPrint) { ... }
⊞TEST(HospitalTest, SortByRatingAndPrint) { ... }
⊞TEST(HospitalTest, PrintDataOutput) { ... }
```

测试结果如下图，可见有部分排序函数中，对 vector<Hospital>的操作没有直接作用到数据库上，所以排序结果与预期不符。这一点可以根据系统的真实用途，按照情况进

行优化。

```
[       OK ] HospitalTest.PrintDataOutput (1 ms)
[----------] 5 tests from HospitalTest (10 ms total)

[----------] Global test environment tear-down
[==========] 5 tests from 1 test case ran. (11 ms total)
[  PASSED  ] 2 tests.
[  FAILED  ] 3 tests, listed below:
[  FAILED  ] HospitalTest.PrintAndSortByBedsPrice
[  FAILED  ] HospitalTest.SortByBedsAvailableAndPrint
[  FAILED  ] HospitalTest.SortByRatingAndPrint
```

四、模糊测试报告

模糊测试 是一种软件测试技术。其核心思想是将自动或半自动生成的随机数据输入到一个程序中，并监视程序异常，如崩溃，断言失败，以发现可能的程序错误，比如内存泄漏。模糊测试常常用于检测软件或计算机系统的安全漏洞。

1. 模糊测试工具的选择与安装

在实验中，我选择了 AFL++作为本次模糊测试实验的工具。
我通过 docker 安装 AFL++后，将自己实现的项目复制进 my_test 文件夹进行模糊测试，测试截图如下：

2. 模糊测试报告

每个项目运行时间表格如下：

项目	时间
项目 1	1 小时 36 分钟
项目 2	58 分钟
自身项目	56 分钟

或许是由于我挑选的开源项目代码量非常少，模糊测试工具没有找到能使项目崩溃的测试用例。

现有的模糊测试工具虽然在软件安全领域中发挥着重要作用，但它们仍存在一些缺陷。其中一个主要问题是测试用例的生成和选择不够智能。模糊测试工具通常依赖于预定义的或随机生成的测试用例来模拟异常输入，但这种方法可能无法有效地识别和利用软件中的潜在漏洞。另外，模糊测试工具的测试效果也受到测试用例的质量和数量的影响，而现有的工具在生成和选择测试用例方面还有待提高。此外，模糊测试工具的运行时间较长，可能会延长软件开发和测试的周期，增加测试成本。

五、程序修复报告

大语言模型在代码错误修复中起到了重要的作用。它可以识别代码中的错误，提供可能的修复方案，并帮助开发者理解代码的意图。在实验中，我利用大语言模型 Chatglm, 尝试修复在本次实验中发现的，项目 1 函数无法处理异常情况，以及自身项目无法改变原来 vector 数据的缺陷。

1. 项目 Json-parser 中，修复__parse_json_unicode 对边界情况的处理：

在函数体中，有部分情况 unicode 无法转换成 utf-8 编码，但函数体没有对这些情况加以排除。

```
static int __parse_json_unicode(const char *cursor, const char **end,
                                char *utf8)
{
    unsigned int code;
    unsigned int next;
    int ret;

    ret = __parse_json_hex4(cursor, end, &code);
    if (ret < 0)
        return ret;

    if (code >= 0xdc00 && code <= 0xdfff)
        return -2;

    if (code >= 0xd800 && code <= 0xdbff) { ... }

    if (code <= 0x7f) { ... }
    else if (code <= 0x7ff) { ... }
    else if (code <= 0xffff) { ... }
    else { ... }
}
```

我向 chatglm 提供了这个函数，并详细描述了问题，得到回复结果如下，大语言模型反而删去了其中一个错误码的范围处理，这与我准备修复的错误相悖：

```
static int __parse_json_unicode(const char* cursor, const char** end,
    char* utf8)
{
    unsigned int code;
    unsigned int next;
    int ret;
    ret = __parse_json_hex4(cursor, end, &code);
    if (ret < 0)
        return ret;
    // Check if the code is a high surrogate (D800 - DBFF)
    if (code >= 0xD800 && code <= 0xDBFF) { ... }
    // Handle the UTF-8 encoding of the Unicode character
    if (code <= 0x7f) { ... }
    else if (code <= 0x7ff) { ... }
    else if (code <= 0xffff) { ... }
    else { ... }
}
```

最后，我给出的修复方案如下，我增添了需要报错解析码范围。

```
static int __parse_json_unicode(const char* cursor, const char** end,
    char* utf8)
{
    unsigned int code;
    unsigned int next;
    int ret;
    ret = __parse_json_hex4(cursor, end, &code);
    if (ret < 0)
        return ret;
    if (code > 0x10FF) {
        return -2;
    }
```

2. 针对我自己的项目，我想修复函数无法更改原始 vector<>值的问题，于是向模型提供了源代码和错误：

```
void SortByBedsPrice(
    vector<Hospital> hospitals)
{
    sort(hospitals.begin(),
         hospitals.end(),
         beds_price);

    cout << "SORT BY Available Beds Price:"
         << endl
         << endl;

    PrintHospitalData(hospitals);
}
```

得到的回答如下：

```
// Function to sort the hospital data by Bed Price
void SortByBedsPrice(std::vector<Hospital>& hospitals) {
    std::sort(hospitals.begin(), hospitals.end(), [](const Hospital& a, const Hospita
        return a.price < b.price;
    });

    std::cout << "SORT BY Available Beds Price:" << std::endl << std::endl;
    PrintHospitalData(hospitals);
}
```

经过测试，这样修改的确改变了 vector<Hospital>的值，达到了我希望修复的效果。在实验中，我意识到，大语言模型可能无法完全理解复杂的代码结构和上下文，导致错误的修复建议。其次，它可能无法处理一些特殊的错误情况，如语法错误或类型错误。此外，它也无法完全替代人工的代码审查和测试，因为机器模型可能无法完全理解人类的编码习惯和逻辑。因此，尽管大语言模型在代码错误修复中具有一定的作用，但仍需要谨慎使用，并结合人工审查和测试来确保代码的质量。

实验评分：_____

指导教师签字：_____

年 月 日