finix Documentation

发布 *1.0.0*

teststars

2020 年 09 月 25 日

Contents

CHAPTER 1

测试用例自动生成技术的意义

1.1 传统的 UI 自动化测试技术的缺点

自动化测试最大的挑战就是需求的变化，而自动化脚本本身就需要修改、扩展、debug，去适应新的功能，如果投入产出比太低，那么自动化测试也失去了其价值和意义。另外自动化仍然属于黑盒范围，无法对白盒覆盖等技术有有效的提升。

1.2 测试左移后单元测试（程序级自动化）面临的问题

编写单元测试用例耗时长传统程序级测试用例的编写会耗费开发人员大量的工时，比如 TDD 测试驱动开发里面的单元测试无法有效实施，导致所有测试几乎全部依赖于系统级黑盒测试。程序级测试用例的开发成本是功能实现代码本身时间至少为 1：1，绝大部分企业选择放弃开发程序级测试，而采用系统级测试方法。

需求变更、持续投入大需求发生变化导致程序实现发生变化后，程序集用例也需要发生变化，和自动化面临的问题一样，单元测试本身的可维护性问题导致投入是持续的而不是一次性的，会打消其企业应用的热情。

适应性不全面很多单元在未组装成系统的时候切入，如果需要进行测试需要进行大量的 mock 操作或者桩模拟，这个过程会造成单元测试的不精确性。

测试数据准备过程长程序集测试数据量很大，全部需要用户来进行准备无法全自动从前序系统测试过程中拿到。

1.3 编写单元测试的一般流程

首先编写被测函数的驱动函数，驱动函数主要功能为针对被测函数进行赋值

构造被测函数的所有参数值

构造被测函数中使用的全局变量，并赋值

获取对被测函数的返回值

校验，判断是否针对输入得到预期输出

1.4 Case

See http://www.codewings.net/tutorials/

1.5 Download

See http://www.codewings.net/download/

1.6 Support

See http://www.codewings.net/contact/

CHAPTER 2

wings 单元测试自动生成技术

2.1 程序级测试用例自动生成技术的特性

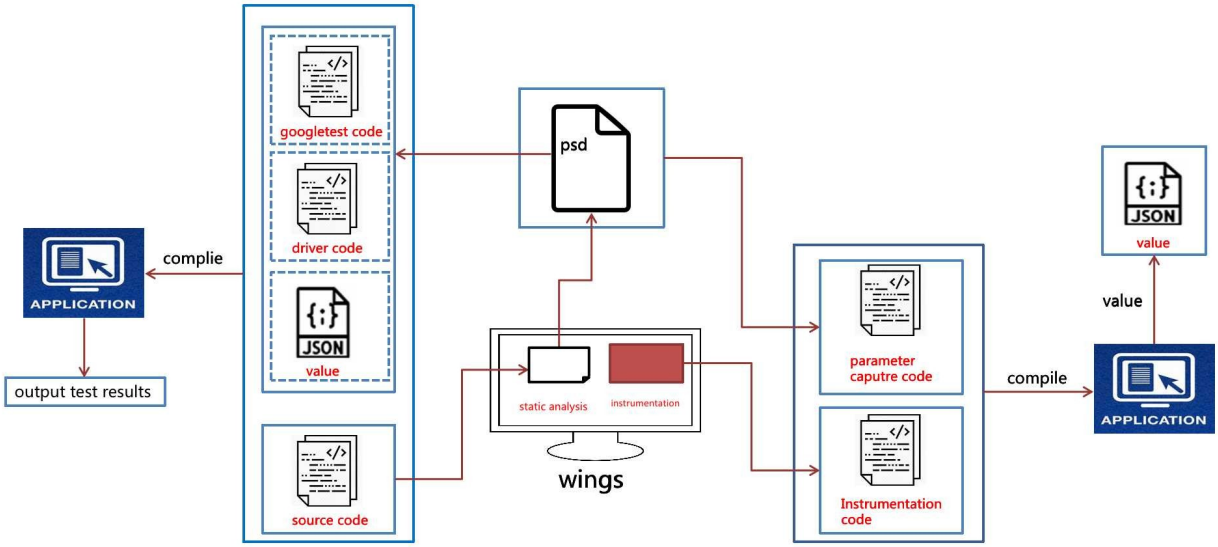
- Wings 是智能的全自动测试用例驱动生成系统，可以将任意复杂参数结构逐步分解为基本数据类型，并全自动生成测试驱动程序和参数捕获程序。
- Wings 支持面向过程 C 语言以及面向对象 C++ 语言的驱动生成和参数捕获，能够支持高级语言特性和基本的内置类型，支持被测函数运行必要的全局变量的构建和赋值。
- 支持多层次的可视化的数据表格来对变量进行赋值，而无需关注驱动程序本身。数据表格可以表达任意深度和多层次的数据关系，用户只需要对表格数据进行编辑，自动生成的驱动程序，会自动完成表格数据的读取和参数赋值的构造过程。
- 能够区分系统数据类型和用户自定义类型，对于复杂的约定俗成系统类型可由用户自定义扁平式赋值模板，例如 std::vector 类型，而不是把复杂的系统变量全部展开，内部集成常用系统类型的模板

2.2 wings 整体框架图

首先 wings 利用代码静态分析技术，提取被测程序的主干信息，主要步骤如下：

1. 获取不同平台的编译数据库文件
2. 依据编译数据库文件，对被测程序的源代码进行静态分析，将提取到的信息存储与 psd 结构中
3. 读取 psd 结构生成对应的驱动代码、googletest 代码、以及值文件，和被测源代码一同编译，测试输出结果

4. 读取 psd 结构，生成参数捕获的代码，插入被测源文件中进行编译，获取程序运行时的参数值



CHAPTER 3

Paramemter Struture Description(PSD)

PSD 是指对源程序信息进行提取后的描述信息，主要包括类名、命名空间、类的成员变量信息、类的函数信息，以及各种类型信息等。针对类型信息的描述，能够将复杂类型进行展开分解到最基本数据类型（char、int、string 等）。其中 c 语言，以一个文件为单元，提取所有的函数信息，c++ 语言，提取类的所有信息。PSD 结构体存储在 XML 文件中，不同的 XML 文件存储不同的描述信息。

RecordDecl.xml: 存储整个项目工程中的结构体，联合体与类的分析展开结果。

funcPoint.txt：存储函数参数为函数指针的分析结果。

funcCount.txt: 存储分析到的全部函数，参数个数，参数类型信息。

void.txt: 存储函数参数为 void* 的分析结果。

filename.xml：存储 c 语言文件的信息。

针对一些属性简单说明如下：

type 属性

ZOA_CHAR_S/ZOA_UCHAR/ZOA_INT/ZOA_UINT/ZOA_LONG ZOA_ULONG/ZOA_FLOAT/ZOA_UFLOAT/ZOA_SHOTR ZOA_USHORT/ZOA_DOUBLE/ZOA_UDOUBLE	基本类型
StructureOrClassType	结构体类型
ZOA_FUNC	函数指针类型
ZOA_UNION	联合体类型
ZOA_ENUM	枚举类型
ClassType	类类型

basetype 属性

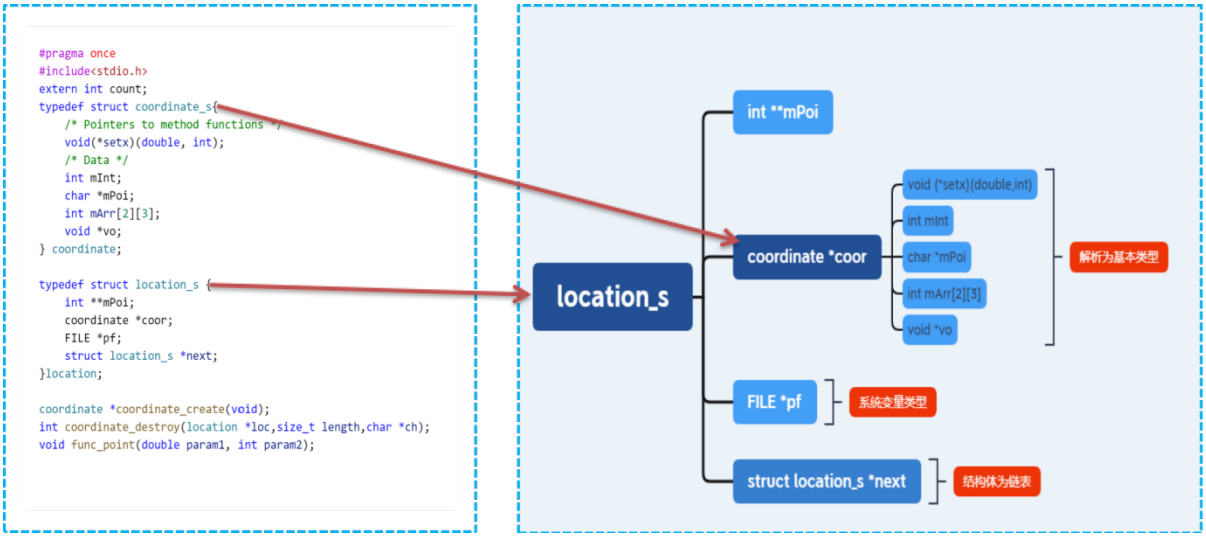
BuiltinType	基本类型
ArrayType	数组类型
PointerType	指针类型
StructureOrClassType	结构体类型
UnionType	联合体类型
EnumType	枚举类型
FunctionPointType	函数指针类型

其他属性

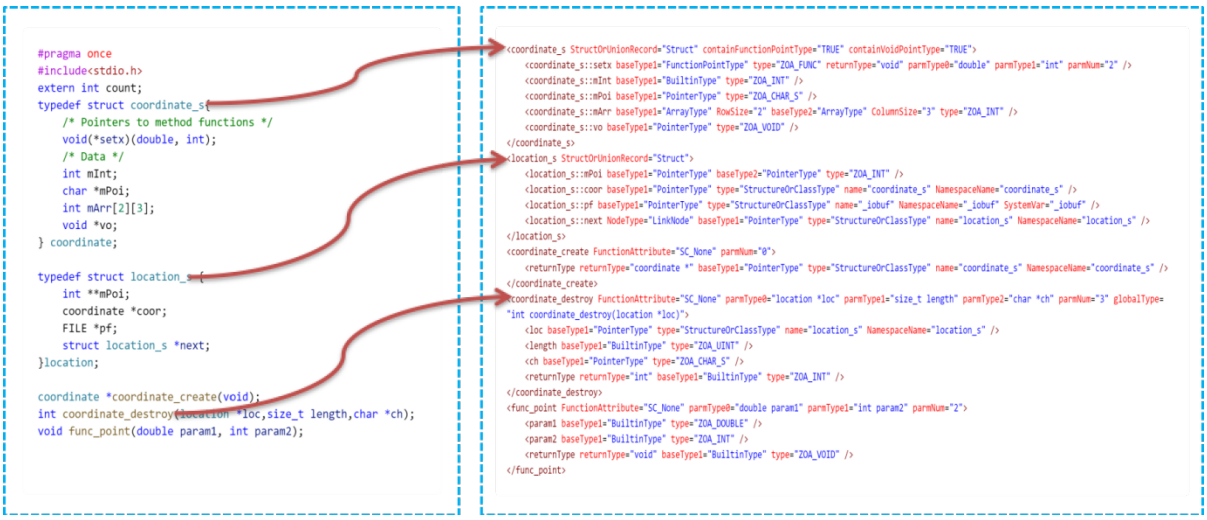
Name	代表结构体、类、联合体名字
NodeType	代表链表类型
parmType	代表函数参数类型
parNum	代表函数参数个数
SystemVar	代表此类型为系统头文件类型
value	代表枚举类型的值
bitfield	代表位域类型所占字节
returnType	代表返回值类型
Field	类成员变量
Method	类构造函数
paramName	类构造函数参数名
paramType	类构造函数参数类型
TemplateArgumentType	STL 结构参数类型
WingsTemplateArgument	STL 结构嵌套参数名字
TemplateArgumentValue	STL 结构中参数为具体值
FunctionModifiers	函数访问权限
FunctionAttribute	函数是 extern 或者 static 函数
FuncClassName	函数所属类
OperatorFundecl	重载运算符函数
Operator	重载运算符类型

3.1 c 语言 psd 结构说明

针对复杂类型，例如结构体类型 location_s，成员变量中除了基本数据类型之外，还存在包含结构体类型的情况，如下图所示的代码中，location_s 中包含 coordinate_s 结构体，以及 FILE 等类型的信息，针对不同的类型进行标记区分。

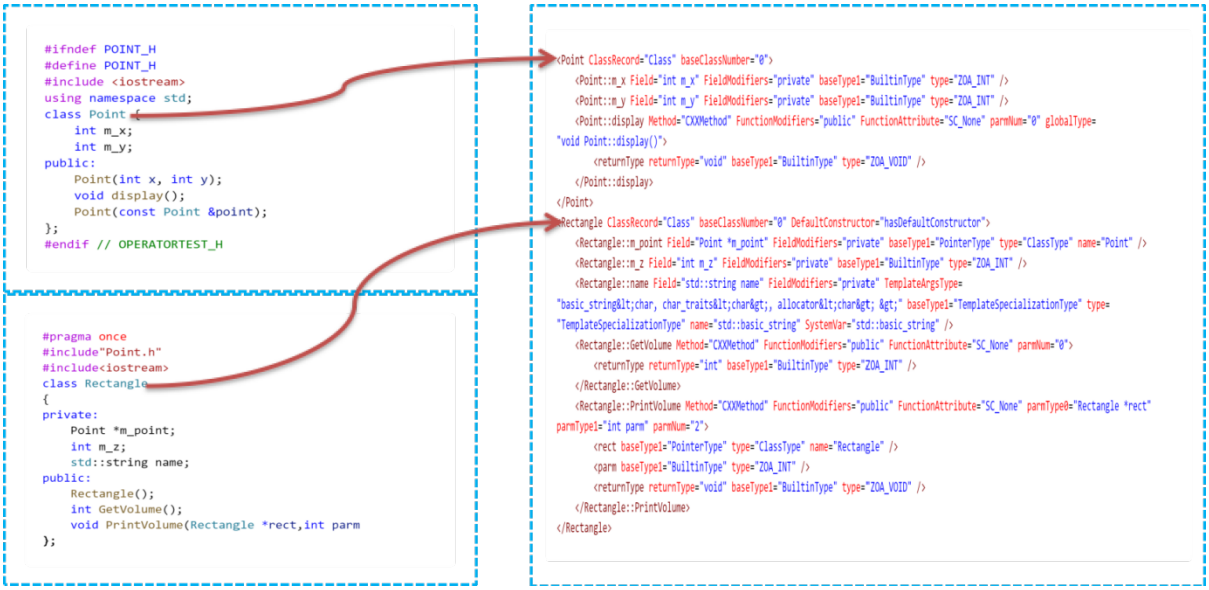


以上图的 psd 存储结构如下图所示，其中结构体的描述信息主要包括，成员变量名、成员变量的类型、以及判断成员变量是否为系统变量或者链表等信息。针对不同的信息，在驱动生成或者参数捕获时，做不同的信息处理。



3.2 c++ psd 结构说明

c++ 的主要表示类型是类，因此测试是 c++ 以一个类为单元做测试，类主要包括类的成员变量名以及类型信息，成员变量的访问权限信息。类的成员函数分为构造函数、内联函数、虚函数等，成员函数的参数信息以及类型信息等。



CHAPTER 4

驱动代码的说明

驱动代码指单元测试代码，wings 针对函数参数的类型，完成自动单元测试代码的编写。

4.1 c 语言驱动代码的说明

wings 利用 PSD 的描述信息，完成单元测试驱动代码的生成。

4.1.1 驱动代码的命名规则

Wings 生成的驱动代码，存储在 drivercode 文件夹中。

注：所有代码的命名规则采用 google c++ 的命名规范，一些稍微特殊的除外。

（1）driver.cc 与 driver.h，主要针对程序中使用到的一些公共函数以及头文件

（2）同一个结构体或者联合体，可能会作为多个函数的参数使用，为了避免代码的重复，wings 针对所有的结构体和联合体的不同类型，封装成不同的驱动函数或者参数捕获函数。driver_structorunion.cc 存储结构体驱动函数的代 driver_structorunion.h 对应的头文件。

举例说明如下：

针对结构体信息，会对应生成下图中不同的驱动函数信息，如下图所示：

```
struct coordinate_s DriverStructcoordinate_s(cJSON *coordinate_sRoot);
struct coordinate_s *DriverStructcoordinate_sPoint(cJSON *coordinate_sRootPoint, int␣
↪row);
struct coordinate_s **DriverStructcoordinate_sPointPoint(cJSON *coordinate_
↪sRootPointPoint, int row, int column);
```

(3) 结构体实现函数的命名规则为：DriverStruct+ 结构体名字 + 类型，其中 Point 代表一级指针或者一维数组，PointPoint 代表二级指针或者二维数组使用。源文件驱动的实现，以每个 c 文件以一个单元，针对每个 c 文件中的每个函数，生成对应的驱动函数

源文件的命名规则为：driver_+ 源文件名 +.cc 例如：driver_nginx.cc

驱动函数的命名规则：Driver_+ 函数名例如：Driver_ngx_show_version_info(void); 注意：针对 static 函数的测试，需要去掉 static 的函数

驱动函数之前添加原函数的声明，例如：extern void ngx_show_version_info(void); Driver_ngx_show_version_info(void);

(4) 返回值的打印输出返回值的打印输出函数命名规则：Driver+Return+Print_+ 函数名。例如：DriverReturnPrint_ngx_show_version_info();

(5) 用户源代码中的 main 函数需要手动注释掉，wings 会在驱动代码中，重新生成一个 main 函数文件，来进行测试。Wings 会生成驱动 main 的主函数文件为:gtest_auto_main.cc 注意：用户依据需要，自行选择使用。

4.1.2 驱动代码的简单说明

(1) Wings 主要针对参数进行逐层展开，解析到最底层为基本类型进行处理。驱动的赋值部分，主要就是针对基本类型进行处理。(注：特殊类型，比如 FILE 等，后面会详细讲解如何赋值) 针对 int 类型，进行说明：

int p; int *p; int **p; int ***p;

int p[1]; int p[2][3]; int p[1][2][3];

int(*p)[]; int(*p)[][3]; int *(*p)[]; int (**p)[];

int *a[]; int **a[]; int *a[][3]; int (*a[])[];

Wings 会针对基本类型的以上 15 中类型，进行不同的赋值。

举例如下：

函数原型：如下图中的 int coordinate_destory (location_s *loc,size_t length,char *ch)

```c
#pragma once
#include<stdio.h>
extern int count;
typedef struct coordinate_s{
    /* Pointers to method functions */
    void(*setx)(double, int);
    /* Data */
    int mInt;
    char *mPoi;
    int mArr[2][3];
    void *vo;
} coordinate;

typedef struct location_s {
    int **mPoi;
    coordinate *coor;
    FILE *pf;
    struct location_s *next;
}location;

coordinate *coordinate_create(void);
int coordinate_destroy(location *loc,size_t length,char *ch);
void func_point(double param1, int param2);
```

其中返回值的类型为 int

第一个参数的类型为 location *

第二个参数的类型为 size_t

第三个参数的类型为 char *

针对以上三个参数进行赋值。

wings 针对 coordinate_destory 函数的完整驱动代码在以下部分进行详细说明.

gtest_auto_main.cc 此文件为调用 gtest 的入口文件，内容如下图所示：

```
#define _CRT_SECURE_NO_WARNINGS
#include "gtest/gtest.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
  testing::InitGoogleTest(&argc, argv); //初始化gtest模块
  RUN_ALL_TESTS();                      //调用各个单元测试函数
  // Start();
  system("pause");
  return 0;
}
```

针对每个测试的.c 文件，会生成对应的 driver_ 文件名 _gtest.cc 文件，针对函数 coordinate_destory 的返回值进行期望对比操作。

如下图中，获取函数的返回值为 return，用户填写的期望的返回值为 expected。

```
#include "driver_wings_c_demo_coordinates.h"
#include "gtest_struct.h"
/* Gtest Function prototype: */
TEST(wings_c_dem_coordinates, coordinate_destroy) {
  for (int times = 0; times < COORDINATE_DESTROY_TIMES; times++) {
    Drive_coordinate_destroy(times);
    int coordinate_destroy_len = strlen("coordinate_destroy");
    char *coordinate_destroy_sp =
        (char *)malloc(sizeof(char) * (coordinate_destroy_len + 2));
    sprintf(coordinate_destroy_sp, "coordinate_destroy%d", times);
    const char *jsonFile =
        "drivervalue/wings_c_dem_coordinates/coordinate_destroy.json";
    char *wings_c_demo_coordinates_coordinate_destroy_json_data =
        get_json_data(jsonFile);
    cJSON *Root =
        cJSON_Parse
(wings_c_demo_coordinates_coordinate_destroy_json_data);
    cJSON *coordinate_destroy_Root =
        cJSON_GetObjectItem(Root, coordinate_destroy_sp);
    free(coordinate_destroy_sp);
    int _return_actual =
        cJSON_GetObjectItem(coordinate_destroy_Root, "return")->valueint;
    int _return_expected =
        cJSON_GetObjectItem(coordinate_destroy_Root, "return")->valueint;
    /* return_expected */
    EXPECT_EQ(_return_expected, _return_actual);
  }
}
```

下图为被测函数的值文件。

```
{
   "coordinate_destroy_int0" : {
      "ch" : "Po9",
      "count" : 6830,
      "expected" : 10,
      "length" : 6511,
      "loc" : [
         {
            "coor" : [
               {
                  "mArr" : [
                     [ 7347, 9920, 2929 ],
                     [ 917, 3619, 1082 ]
                  ],
                  "mInt" : 6869,
                  "mPoi" : "Pj7",
                  "setx" : null,
                  "vo" : [ 5953, 1098, 2627 ]
               }
            ],
            "mPoi" : [
               [ 1519, 7676, 8151 ],
               [ 7423, 6830, 475 ],
               [ 4811, 2546, 9351 ]
            ],
            "pf" : {
               "wingsParam1":"c:/1.txt",
               "wingsParam2":"w+"
            }
         }
      ],
      "return" : 10
   }
}
```


下图为对应的驱动代码。

```
extern int coordinate_destroy(location *loc, size_t length, char *ch);
int coordinate_destroy_intTimes = 0;
int Drive_coordinate_destroy(int times)
{
    coordinate_destroy_intTimes = times;
    /* Root is the json object of the value file.coordinate_destroy_int_Root is function.
↪coordinate_destroy_int is json object.  */
    const char *jsonFile = "../drivervalue/wings_c_demo_coordinates/coordinate_destroy_
↪int.json";
    char *json_data = get_json_data(jsonFile);
    cJSON *Root = cJSON_Parse(json_data);
    int coordinate_destroy_int_len = strlen("coordinate_destroy_int");
    char *coordinate_destroy_int_sp = (char *)malloc(sizeof(char) * (coordinate_destroy_
↪int_len + 3));
    sprintf(coordinate_destroy_int_sp, "coordinate_destroy_int%d", times);
    cJSON *coordinate_destroy_int_Root = cJSON_GetObjectItem(Root, coordinate_destroy_
↪int_sp);
    free(coordinate_destroy_int_sp);
    /*It is the 1 global variable: count    coordinate_destroy */
    int _count = cJSON_GetObjectItem(coordinate_destroy_int_Root, "count")->valueint;
    count = _count;
```

(下页继续)

(续上页)

```
    /*It is the 1 parameter: loc    coordinate_destroy*/
    cJSON *loc_Arr_Root = cJSON_GetObjectItem(coordinate_destroy_int_Root, "loc");
    int loc_size = cJSON_GetArraySize(loc_Arr_Root);
    struct location_s *_loc = DriverStructlocation_sPoint(loc_Arr_Root, loc_size);
    /*It is the 2 parameter: length    coordinate_destroy*/
    unsigned int _length = (unsigned int)cJSON_GetObjectItem(coordinate_destroy_int_Root,
↪ "length")->valueint;
    /*It is the 3 parameter: ch    coordinate_destroy*/
    char *_ch;
    {
        char *_ch_str = cJSON_GetObjectItem(coordinate_destroy_int_Root, "ch")->
↪valuestring;
        _ch = (char *)malloc(sizeof(char) * (strlen(_ch_str) + 1));
        memcpy(_ch, _ch_str, strlen(_ch_str));
        _ch[strlen(_ch_str)] = '\0';
    }
    //Function Call
    int returnType = coordinate_destroy(_loc, _length, _ch);
    return 0;
}
```

```
struct location_s *DriverStructlocation_sPoint(cJSON *location_sRootPoint, int row)
{
    struct location_s *_location_s = (struct location_s *)malloc(sizeof(struct location_
↪s) * row);
    for (int i = 0; i < row; i++)
    {
        cJSON *location_s_Root = cJSON_GetArrayItem(location_sRootPoint, i);

        int **_mPoi;
        cJSON *mPoi_Root = cJSON_GetObjectItem(location_s_Root, "mPoi");
        int mPoi_row = cJSON_GetArraySize(mPoi_Root);
        _mPoi = (int **)malloc(sizeof(int *) * mPoi_row);
        for (int i = 0; i < mPoi_row; i++)
        {
            cJSON *mPoi_Root_Row = cJSON_GetArrayItem(mPoi_Root, i);
            int mPoi_column = cJSON_GetArraySize(mPoi_Root_Row);
            _mPoi[i] = (int *)malloc(sizeof(int) * mPoi_column);
            for (int j = 0; j < mPoi_column; j++)
```

(下页继续)

(续上页)

```
            {
                _mPoi[i][j] = cJSON_GetArrayItem(mPoi_Root_Row, j)->valueint;
            }
        }

        _location_s[i].mPoi = _mPoi;
        cJSON *coor_Arr_Root = cJSON_GetObjectItem(location_s_Root, "coor");

        int coor_size = cJSON_GetArraySize(coor_Arr_Root);
        struct coordinate_s *_coor = DriverStructcoordinate_sPoint(coor_Arr_Root, coor_
↪size);

        _location_s[i].coor = _coor;
        cJSON *pf_Arr_Root = cJSON_GetObjectItem(location_s_Root, "pf");
        cJSON *pf_Root = cJSON_GetArrayItem(pf_Arr_Root, 0);

        /* wingsParam1 */
        unsigned char *_wingsParam1;
        {
            char *_wingsParam1_str = cJSON_GetObjectItem(pf_Root, "wingsParam1")->
↪valuestring;
            _wingsParam1 = (unsigned char *)malloc(sizeof(unsigned char) * (strlen(_
↪wingsParam1_str) + 1));
            memcpy(_wingsParam1, (unsigned char *)_wingsParam1_str, strlen(_wingsParam1_
↪str));
            _wingsParam1[strlen(_wingsParam1_str)] = '\0';
        }

        /* wingsParam2 */
        unsigned char *_wingsParam2;
        {
            char *_wingsParam2_str = cJSON_GetObjectItem(pf_Root, "wingsParam2")->
↪valuestring;
            _wingsParam2 = (unsigned char *)malloc(sizeof(unsigned char) * (strlen(_
↪wingsParam2_str) + 1));
            memcpy(_wingsParam2, (unsigned char *)_wingsParam2_str, strlen(_wingsParam2_
↪str));
            _wingsParam2[strlen(_wingsParam2_str)] = '\0';
        }
        struct _iobuf *_pf = _iobufFunctionPointer(_wingsParam1, _wingsParam2);
```

(下页继续)

(续上页)

```
        _location_s[i].pf = _pf;
        cJSON *next_Arr_Root = cJSON_GetObjectItem(location_s_Root, "next");

        int next_size = cJSON_GetArraySize(next_Arr_Root);
        struct location_s *_next = DriverStructlocation_sPoint(next_Arr_Root, next_size);

        _location_s[i].next = _next;
    }
    return _location_s;
}
```

4.2 c 版本参数捕获代码说明

wings 参数捕获功能主要是在运行时获取函数的参数值，全局变量值以及返回值的信息

4.2.1 参数捕获代码的命名规则

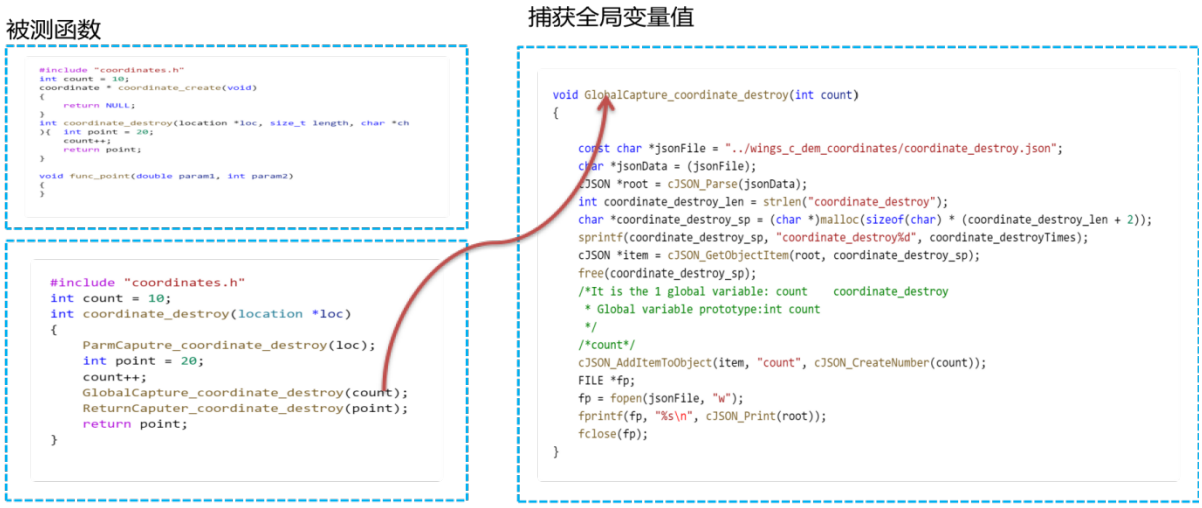
wings 的参数捕获代码存储在 paramcaputrecode 文件夹中。其中命名规则同驱动格式一样，将所有的 driver 替换为 param 即可。

4.2.2 参数捕获代码的示例

如下所示，针对函数 coordinate_destroy 中，会自动生成捕获参数、全局变量以及返回值的信息。

```
#ifndef _PARAMCAPTURE_WINGS_C_DEMO_COORDINATES_H_
#define _PARAMCAPTURE_WINGS_C_DEMO_COORDINATES_H_
#include "ParamCapture.h"
#include "ParamCapture_structorunion.h"
void ReturnCapture_coordinate_create(coordinate_s returnType);

void ParamCapture_coordinate_destroy(location *loc, size_t length, char *ch);
void GlobalCapture_coordinate_destroy(int count);
void ReturnCapture_coordinate_destroy(int returnType);

void ParamCapture_func_point(double param1, int param2);
void ReturnCapture_func_point(void returnType);
#endif // WINGS_C_DEMO_COORDINATES
```

下图将展示捕获全局变量的自动生成代码。



4.3 c++ 驱动代码

c++ 中主要是针对每个类进行测试，wings 会自动生成每个类的驱动代码以及对应的 gtest 代码。

4.3.1 c++ 驱动代码的命名规则

每个类生成驱动对应的文件名为：driver+ 类名.cc 以及.h，驱动类的名字为：Driver+ClassName(类名)，c++ 中存在很多运算符重载的函数，为了保证函数的唯一性，针对类的函数从 0 开始进行编号处理，即 Driver+ 函数名 + 编号。

4.3.2 c++ 驱动代码的详细介绍

c++ 中会针对每个类生成对应的测试类代码，如下所示，Rectangle 为被测试类，包含 3 个成员变量的类型分别为 Point、int 、std::string, 假设 Rectangle 作为函数参数，则此类的对象构造需要对 3 个成员变量进行赋值操作。

```
#include "Rectangle.h"
#include "driver.h"
class DriverRectangle {
public:
  DriverRectangle(Json::Value Root, int times);
  ~DriverRectangle();
  int DriverRectangleGetVolume0(int times);
  void ReturnDriver_GetVolume0(int returnType);
  int GetVolume0Times;
```

(下页继续)

(续上页)

```
  int DriverRectanglePrintVolume1(int times);
  int PrintVolume1Times;
private:
  Rectangle *_Rectangle;
};
```

wings 会自动针对每个类在源代码中插入一个构造函数如下所示，对成员变量进行赋值操作。

```
class Rectangle
{
private:
        Point *m_point;
        int m_z;
        std::string name;
public:
        Rectangle();
        int GetVolume();
        void PrintVolume(Rectangle *rect, int parm);
public:
        Rectangle(Point *m_point, int m_z, std::string name, bool wings)
        {
                //LOGI("Rectangle::Rectangle");
                this->m_point = m_point;
                this->m_z = m_z;
                this->name = name;
        }
};
```

针对每个驱动类，对应的会生成一个构造函数，来初始化被测试类，如下所示：

```
DriverRectangle::DriverRectangle(Json::Value Root, int times) {
  Json::Value Rectangle_Root = Root["Rectangle" + to_string(times)];
  int pointSize = 0;
  Json::Value m_point_Root = Rectangle_Root["m_point"][pointSize];
  /* m_x */
  int _m_point_m_x = m_point_Root["m_x"].asInt();
  /* m_y */
  int _m_point_m_y = m_point_Root["m_y"].asInt();
  Point *_m_point = new Point(_m_point_m_x, _m_point_m_y, false);
  /* m_z */
  int _m_z = Rectangle_Root["m_z"].asInt();
```

(下页继续)

(续上页)

```
  string _name = Rectangle_Root["name"].asString();
  _Rectangle = new Rectangle(_m_point, _m_z, _name, false);
}
DriverRectangle::~DriverRectangle() {
  if (_Rectangle != nullptr) {
        delete _Rectangle;
  }
}
```

驱动类中，针对每个函数生成一个驱动函数。如下所示：

```
int DriverRectangle::DriverRectanglePrintVolume1(int times) {
  PrintVolume1Times = times;
  /* Root is the json object of the value file.PrintVolume1_Root is
   * function.PrintVolume1 is json object.  */
  const char *jsonFilePath = "drivervalue/Rectangle/PrintVolume1.json";
  Json::Value Root;
  Json::Reader _reader;
  ifstream _ifs(jsonFilePath);
  _reader.parse(_ifs, Root);
  Json::Value PrintVolume1_Root = Root["PrintVolume1" + to_string(times)];
  /*It is the 1 parameter: rect    PrintVolume1*/
  int pointSize = 0;
  Json::Value rect_Root = PrintVolume1_Root["rect"][pointSize];
  int pointSize = 0;
  Json::Value m_point_Root = rect_Root["m_point"][pointSize];
  /* m_x */
  int _rect_m_point_m_x = m_point_Root["m_x"].asInt();
  /* m_y */
  int _rect_m_point_m_y = m_point_Root["m_y"].asInt();
  Point *_rect_m_point = new Point(_rect_m_point_m_x, _rect_m_point_m_y, false);
  /* m_z */
  int _rect_m_z = rect_Root["m_z"].asInt();
  string _rect_name = rect_Root["name"].asString();
  Rectangle *_rect = new Rectangle(_rect_m_point, _rect_m_z, _rect_name, false);
  /*It is the 2 parameter: parm    PrintVolume1*/
  int _parm = PrintVolume1_Root["parm"].asInt();
  // The Function of Class    Call
  _Rectangle->PrintVolume(_rect, _parm);
  return 0;
```

(下页继续)

(续上页)

```
}
```

每个驱动类，对应一个 gtest 调用类，来进行期望的对比，如下所示

```
#pragma once
#include"Point.h"
#include<iostream>
class Rectangle
{
private:
    Point *m_point;
    int m_z;
    std::string name;
public:
    Rectangle();
    int GetVolume();
    void PrintVolume(Rectangle *rect,int parm
);
```

```
#define _SILENCE_TR1_NAMESPACE_DEPRECATION_WARNING 1
#include "driverRectangle.h"
#include "gtest/gtest.h"
class GtestRectangle : public testing::Test {
protected:
  virtual void SetUp() {
    const char *jsonFilePath = "../drivervalue/RecordDecl.json";
    Json::Value Root;
    Json::Reader _reader;
    ifstream _ifs(jsonFilePath);
    _reader.parse(_ifs, Root);
    driverRectangle = new DriverRectangle(Root, 0);
  }
  virtual void TearDown() {
    if(driverRectangle!=nullptr)
    {
        delete driverRectangle;
    }
  }
  DriverRectangle *driverRectangle;
};
```

gtest 函数的对比如下图所示：

```
#pragma once
#include"Point.h"
#include<iostream>
class Rectangle
{
private:
    Point *m_point;
    int m_z;
    std::string name;
public:
    Rectangle();
    int GetVolume();
    void PrintVolume(Rectangle *rect,int parm
);
```

```
#define _SILENCE_TR1_NAMESPACE_DEPRECATION_WARNING 1
#include "driverRectangle.h"
#include "gtest/gtest.h"
class GtestRectangle : public testing::Test {
protected:
  virtual void SetUp() {
    const char *jsonFilePath = "../drivervalue/RecordDecl.json";
    Json::Value Root;
    Json::Reader _reader;
    ifstream _ifs(jsonFilePath);
    _reader.parse(_ifs, Root);
    driverRectangle = new DriverRectangle(Root, 0);
  }
  virtual void TearDown() {
    if(driverRectangle!=nullptr)
    {
        delete driverRectangle;
    }
  }
  DriverRectangle *driverRectangle;
};
```

4.4 c++ 参数捕获代码的说明

4.4.1 c++ 参数捕获代码的命名规则

c++ 的参数捕获的命名规则与驱动一样，将 driver 换成 paramcaputre 即可。

4.4.2 c++ 参数捕获代码的说明

针对每个类，生成一个参数捕获的类，捕获类中分别对应每个函数会生成捕获参数、全局变量、以及返回值的函数。

```
#pragma once
#include "paramcapture.h"
```

(下页继续)

(续上页)

```
class ParamCaptureRectangle
{
        public:
        ParamCaptureRectangle();
         ~ParamCaptureRectangle();

        void ParamCapture_GetVolume0();
        void GlobalCapture_GetVolume0();
        void ReturnCapture_GetVolume0(int returnType);

        void ParamCapture_PrintVolume1(Rectangle *rect,int parm);
        void GlobalCapture_PrintVolume1();
        void ReturnCapture_PrintVolume1();
};
```

如何捕获类的成员变量，会在类中插入一个捕获函数，来获取类的私有成员变量，如下所示：

```
class Rectangle
{
private:
        Point *m_point;
        int m_z;
        std::string name;
public:
        Rectangle();
        int GetVolume();
        void PrintVolume(Rectangle *rect, int parm);
public:
        Rectangle(Point *m_point, int m_z, std::string name, bool wings)
        {
                this->m_point = m_point;
                this->m_z = m_z;
                this->name = name;
        }
        Json::Value W_MemberVarCaputre()
        {
                Json::Value Rectangle_Root;
                Rectangle_Root["m_point"]=m_point->W_MemberVarCaputre();
                Rectangle_Root["m_z"]=Json::Value(m_z);
                Rectangle_Root["name"]=Json::Value(name);
```

(下页继续)

(续上页)

```
                return Rectangle_Root;
        }
};
```

CHAPTER 5

特殊赋值类型处理

在处理不同类型的时候，会遇到一些特殊操作的类型，例如 void*、函数指针、系统类型等特殊的类型，针对这些特殊的类型，wings 提供了不同的操作来进行特殊处理。

5.1 函数参数为 void *与函数指针

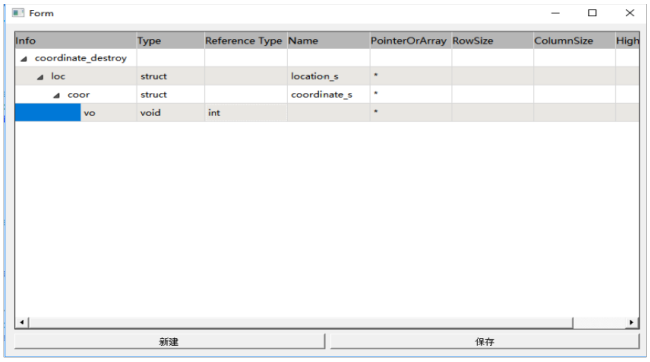
针对函数参数为 void* 与函数指针的类型，wings 首先会利用静态分析技术，获取函数参数为 void *与函数指针时的具体赋值类型。例如下所示函数：

```
void func(void *p);
callFunc(int *p);
int callFunc(int *p)
{
  char *s ="abc";
  func(s);
  return 0;
}
int func(int(*f)(int));
void functest();
void functest()
{
 func(fun);
}
```

Wings 在静态解析时，会分析到 func 在被调用处的赋值类型为 char *，赋值过程中将会对 func 的参数赋值为 char *，wings 在数据表格界面会标记 void *处的具体赋值类型。Wings 在静态分析过程中，会解析到 func 在被调用出的赋值函数为 fun，在赋值过程中将会对 func 的参数赋值为 fun。

针对一些利用分析技术无法确定的类型，wings 会将所有有关的函数展示在界面上，由用户自己选择需要的类型，驱动会处理相对应的类型。如下图所示：

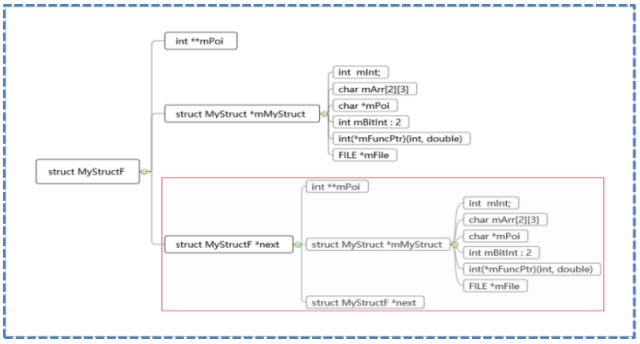
```
#pragma once
#include<stdio.h>
extern int count;
typedef struct coordinate_s{
    /* Pointers to method functions */
    void(*setx)(double, int);
    /* Data */
    int mInt;
    char *mPoi;
    int mArr[2][3];
    void *vo;
} coordinate;

typedef struct location_s {
    int **mPoi;
    coordinate *coor;
    FILE *pf;
    struct location_s *next;
}location;

coordinate *coordinate_create(void);
int coordinate_destroy(location *loc,size_t length,char *ch);
void func_point(double param1, int param2);
```

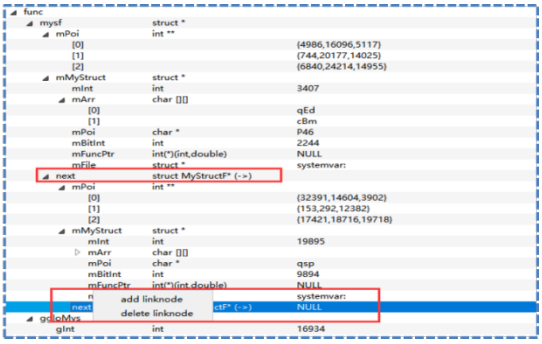


➢ 针对void *类型和函数指针，首先我们会检索到所有与void*与函数指针有关的函数，将信息显示在界面上，由用户进行信息确认，wings会依据配置的信息，生成对应的驱动信息。

5.2 结构体链表

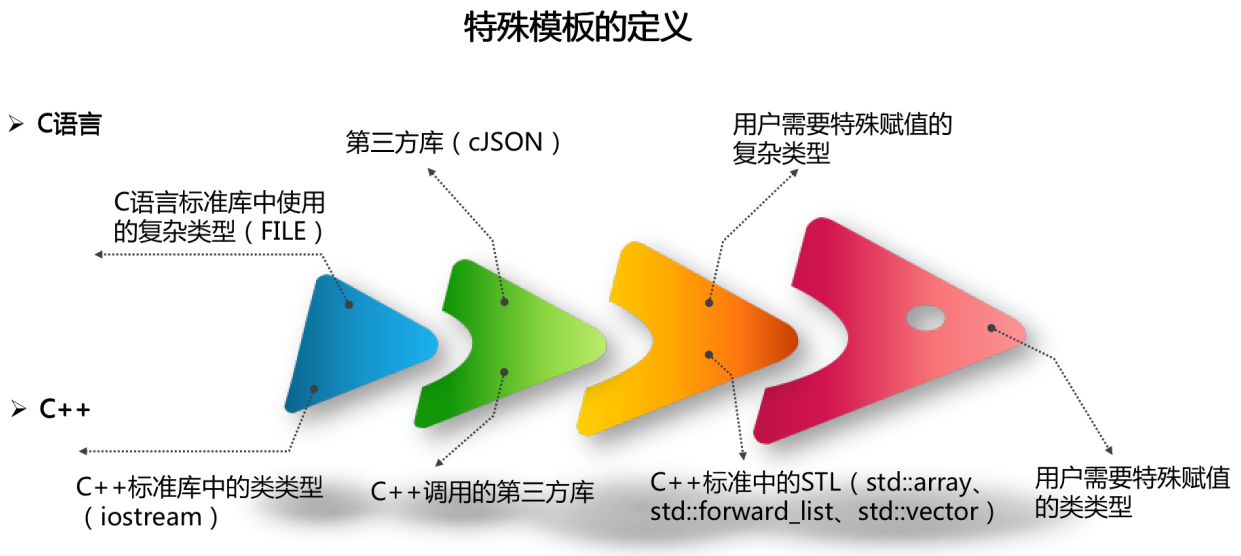
针对链表类型，采用比较灵活的赋值方式，考虑到实际应用中的一些因素，我们针对链表类型，默认赋值两层结构，在实际测试过程中，用户可依据需要自动添加节点。





5.3 结构体、类为系统变量

wings 针对一些 c++ 标准库的头文件，以及一些第三方库的，做特殊模版处理。下图中，是一些特殊系统变量的定义。



举例说明

假设遇到特殊类型的赋值，比如下图中的 sockaddr_in, 首先讲此类型标记为系统头文件中的类型，需要特殊处理。

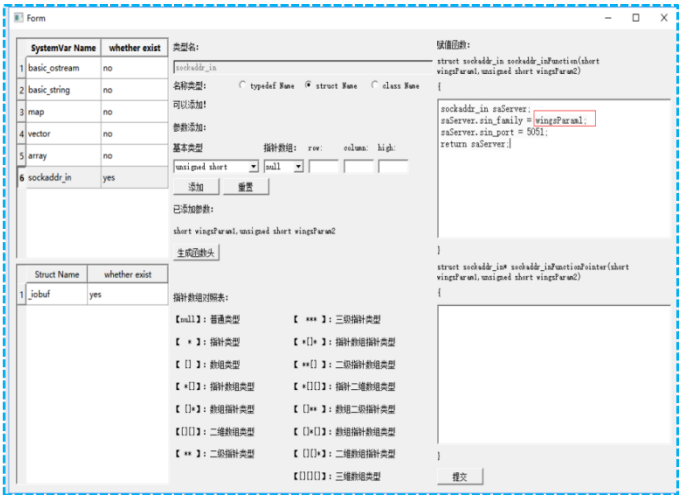
```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

```
#pragma once
#include <iostream>
#include <winsock.h>
#pragma comment(lib,"ws2_32.lib")
class SocketDemo
{
    void creatSocket(struct sockaddr_in saServer);

};
```

sockaddr_in 中成员变量需要用户自定义赋值 * 首先 wings 检测到 sockaddr_in 需要特殊处理，会将此变量显示在模板类的界面

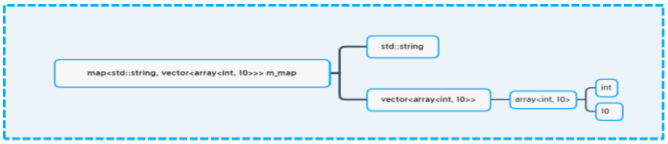
- 然后针对需要特殊处理的变量，例如 sin_family 等，需要用户针对需要的类型进行配置。
- 配置完之后，返回对应的 sockaddr_in 的结构体对象即可。
- 生成驱动代码的时候，会调用用户编写的函数，而需要填写的变量，例如 sin_family 值有我们自动生成。

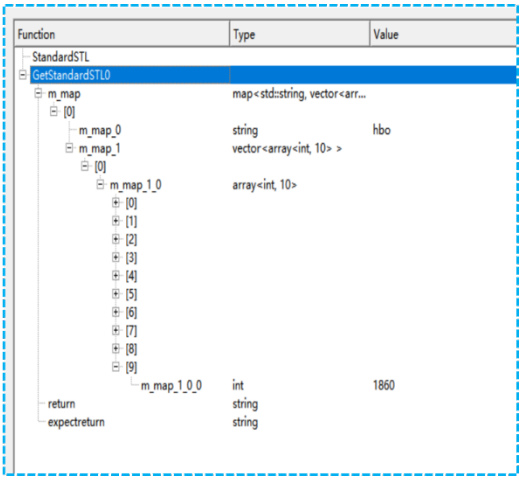
wings 针对特殊模版，有简单的界面进行配置处理即可。



5.4 stl 标准模版库

针对 c++ 中的标准容器，我们将容器中的类型进行展开赋值，默认生成一组值，可以依据需要，在界面上点击进行添加即可。





5.5 c++ 自定义模版类

针对参数为自定义的模板类型，我们会分析模板类的具体类型，例如下图中 GetTest 中模板类的类型为 int 与 double，GetTestDemo 中类型为 std::string 与 double，我们针对模板类同样插入一个构造函数 CustomTemplateClass，实际构造模板类对象的时，调用具体的赋值类型进行构造。

```cpp
#pragma once
template<typename T,typename N>
class CustomTemplateClass
{
public:
    CustomTemplateClass(T m_t,N *m_N,bool wings);
private:
    T m_T;
    N *m_N;
};
template<typename T, typename N>
CustomTemplateClass<T, N>::CustomTemplateClass(T m_t, N *m_n,
bool wings)
{
    this->m_T = m_t;
    this->m_N = m_n;
}
```

```cpp
#pragma once
#include"CustomTemplateClass.hpp"
#include<string>
class TemplateClassTest
{
public:
    TemplateClassTest();
    ~TemplateClassTest();
    void GetTest(CustomTemplateClass<int, double> cum);
    void GetTestDemo(CustomTemplateClass<std::string, double> cum1
);
private:

};
```

CHAPTER 6

参数捕获测试 Demo

6.1 前言

参数捕获是通过在用户源码中插装 wings 生成的参数捕获代码进行获取函数的参数数据，对于基本类型在获取到数据就可以直接存放入 json 文件中，用于后续驱动代码的读取；对于类、结构体等特殊数据类型，我们选择的是将类和结构体的内部数据进行统一处理，生成对应类的 W_MemberVarCaputer() 以及结构体的驱动函数，对应生成两个结构体和类的文件 WingsClassCapture 和 paramcapture_structorunion。PS：当遇到自定义模板类时，上述类的方法不可行，所以自定义模板类是通过直接插装在源代码的头文件中，然后在参数捕获代码中进行调用，当自定义模板类中存在类类型、结构体等复杂数据类型时暂不支持，因为会造成头文件循环包含问题。

6.2 自定义模板类型的参数捕获

6.2.1 测试 demo

```
#include"json/json.h"
#pragma once
#pragma warning(disable:4996)
#include <iostream>
#include <fstream>
using namespace std;
```

(下页继续)

(续上页)

```
namespace Test
{
        struct student
        {
                int a;
                char b;
                string str;
};
        class Teststr
        {
        private:
                int a;
        public:
};
        template<typename T1, typename T2>
        class TestTemplate
        {
        private:
                T1 *t1;
                T2 t2;
                int a;
                char b;
                string s;
                student stu;
public:
//此函数为插装进入的类参数捕获函数
        Json::Value W_MemberVarCaputer()
        {
                Json::Value TestTemplate_Root;
                /*t1*/
                TestTemplate_Root["t1"] = Json::Value(t1);
                /*t2*/
                TestTemplate_Root["t2"] = Json::Value(t2);
                /*a*/
                TestTemplate_Root["a"] = Json::Value(a);
                /*b*/
                char _b[2];
                _b[0] = b;
                _b[1] = '\0';
                TestTemplate_Root["b"] = Json::Value(_b);
```

(下页继续)

(续上页)

```
                TestTemplate_Root["s"] = Json::Value(s);
                /* stu */
                /*Json::Value stu_Root;
                stu_Root = Paramstruct_Test_student(stu_Root, stu);//结构体和类目前无法调
用该函数，会造成头文件循环包含问题
                TestTemplate_Root["stu"] = stu_Root;*/
                return TestTemplate_Root;
}
}
        class ClassTest
        {
        public:
        ClassTest(){};
        ~ClassTest() {};
        void TestTemplateParam(TestTemplate<int, int> TePl);
        void TestTempate(TestTemplate<string, int>  TePo);
        void TestClassNamespace(Teststr te);
        };
}
```

6.2.2 参数捕获代码

```
被测试函数: void TestTemplateParam(TestTemplate<int, int> TePl);
int ParamCaptureTest_ClassTestTestTemplateParam1Times = -1;
void ParamCaptureTest_ClassTest::ParamCapture_TestTemplateParam1(Test::TestTemplate<int,␣
↪int> TePl)
{
        ParamCaptureTest_ClassTestTestTemplateParam1Times++;
        Json::Value Root;
        Json::Value TestTemplateParam1_Root;
        const char* JsonFilePath = "paramcapturevalue/Test_ClassTest/TestTemplateParam1.
↪json";
        if (ParamCaptureTest_ClassTestTestTemplateParam1Times != 0) {
        } else {
                Json::Reader _reader;
                std::ifstream _ifs(JsonFilePath);
                _reader.parse(_ifs, Root);
                TestTemplateParam1_Root = Root["TestTemplateParam1" + std::to_
↪string(ParamCaptureTest_ClassTestTestTemplateParam1Times)];
```

(下页继续)

(续上页)

```
        }

        /*It is the 1 parameter: TePl    TestTemplateParam1
         *
         * Parameters of the prototype:TestTemplate<int, int> TePl
         */

        /* TePl */
        TestTemplateParam1_Root["TePl"] = TePl.W_MemberVarCaputer();

        Root["TestTemplateParam1" + std::to_string(ParamCaptureTest_
↪ClassTestTestTemplateParam1Times)] = TestTemplateParam1_Root;
        std::ofstream JsonFile;
        JsonFile.open(JsonFilePath);
        Json::StyledWriter sw;
        JsonFile << sw.write(Root);
        JsonFile.close();
}
void ParamCaptureTest_ClassTest::GlobalCapture_TestTemplateParam1()
{
        const char* JsonFilePath = "paramcapturevalue/Test_ClassTest/TestTemplateParam1.
↪json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(JsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value TestTemplateParam1_Root = Root["TestTemplateParam1" + std::to_
↪string(ParamCaptureTest_ClassTestTestTemplateParam1Times)];

        Root["TestTemplateParam1" + std::to_string(ParamCaptureTest_
↪ClassTestTestTemplateParam1Times)] = TestTemplateParam1_Root;
        std::ofstream JsonFile;
        Json::StyledWriter sw;
        JsonFile.open(JsonFilePath);
        JsonFile << sw.write(Root);
        JsonFile.close();
}
void ParamCaptureTest_ClassTest::ReturnCapture_TestTemplateParam1()
{
        const char* JsonFilePath = "paramcapturevalue/Test_ClassTest/TestTemplateParam1.
↪json";
```

(下页继续)

(续上页)

```
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(JsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value TestTemplateParam1_Root = Root["TestTemplateParam1" + std::to_
↪string(ParamCaptureTest_ClassTestTestTemplateParam1Times)];

        Root["TestTemplateParam1" + std::to_string(ParamCaptureTest_
↪ClassTestTestTemplateParam1Times)] = TestTemplateParam1_Root;
        std::ofstream JsonFile;
        Json::StyledWriter sw;
        JsonFile.open(JsonFilePath);
        JsonFile << sw.write(Root);
        JsonFile.close();
}
```

6.3 枚举类型的参数捕获

6.3.1 测试 demo

```
#include"json/json.h"
#pragma once
namespace wings_grammars_test {
        enum Code {
                kOk = 0,
                kNotFound = 1,
                kCorruption = 2,
                kNotSupported = 3,
                kInvalidArgument = 4,
                kIOError = 5
        };
        enum class EnumBase
        {
                kNoCompression = 0x0,
                kSnappyCompression = 0x1
        };
        class EnumClassTesting
        {
```

(下页继续)

(续上页)

```
        private:
                EnumBase enumBaseType;
                EnumBase &enumBaseTypeR = enumBaseType;
                Code codeType;
                Code *codePointType;
        public:
                void EnumBaseFunc(wings_grammars_test::EnumBase enumBaseType);
                void EnumBaseFuncR(wings_grammars_test::EnumBase &enumBaseType);
                void CodeFunc(wings_grammars_test::Code codeType);
        public:
        Json::Value W_MemberVarCaputer();
        };
}
```

6.3.2 参数捕获代码

```
对应的测试函数: void CodeFunc(wings_grammars_test::Code codeType);

int ParamCapturewings_grammars_test_EnumClassTestingCodeFunc2Times = -1;
void ParamCapturewings_grammars_test_EnumClassTesting::ParamCapture_CodeFunc2(wings_
↪grammars_test::Code codeType)
{
        ParamCapturewings_grammars_test_EnumClassTestingCodeFunc2Times++;
        Json::Value Root;
        Json::Value CodeFunc2_Root;
        const char* JsonFilePath = "paramcapturevalue/wings_grammars_test_
↪EnumClassTesting/CodeFunc2.json";
        if (ParamCapturewings_grammars_test_EnumClassTestingCodeFunc2Times != 0) {
        }
        else {
                Json::Reader _reader;
                std::ifstream _ifs(JsonFilePath);
                _reader.parse(_ifs, Root);
                CodeFunc2_Root = Root["CodeFunc2" + std::to_string(ParamCapturewings_
↪grammars_test_EnumClassTestingCodeFunc2Times)];
        }
        /*It is the 1 parameter: codeType    CodeFunc2
         *
         * Parameters of the prototype:wings_grammars_test::Code codeType
```

(下页继续)

(续上页)

```
        */
         /*codeType*/
        string codeType_enum;
        if (codeType == wings_grammars_test::Code::kOk) {
                codeType_enum = "kOk";
        }
        if (codeType == wings_grammars_test::Code::kNotFound) {
                codeType_enum = "kNotFound";
        }
        if (codeType == wings_grammars_test::Code::kCorruption) {
                codeType_enum = "kCorruption";
        }
        if (codeType == wings_grammars_test::Code::kNotSupported) {
                codeType_enum = "kNotSupported";
        }
        if (codeType == wings_grammars_test::Code::kInvalidArgument) {
                codeType_enum = "kInvalidArgument";
        }
        if (codeType == wings_grammars_test::Code::kIOError) {
                codeType_enum = "kIOError";
        }
        CodeFunc2_Root["codeType"] = Json::Value(codeType_enum);
        Root["CodeFunc2" + std::to_string(ParamCapturewings_grammars_test_
↪EnumClassTestingCodeFunc2Times)] = CodeFunc2_Root;
        std::ofstream JsonFile;
        JsonFile.open(JsonFilePath);
        Json::StyledWriter sw;
        JsonFile << sw.write(Root);
        JsonFile.close();
}
```

6.3.3 参数捕获代码解析

枚举一般分为普通枚举和强枚举类型（c++11 之后），对于两种枚举的详细区别可自行学习，在此简述，强枚举类型定义的变量只能使用该枚举类型去赋值，而普通枚举值可以简单的理解为 int 型，是可以给其赋 int 型值的；针对这种情况，我们是通过对枚举类型赋值字符串的形式，然后在对应参数不好中给其添加对应枚举类型的前缀（即达到给对应枚举类型赋值的目的）。

6.4 STL 标准库容器参数捕获

6.4.1 测试 demo

```
#include"json/json.h"
#pragma warning(disable:4996)
#pragma once
#include <iostream>
#include <vector>
#include<map>
#include <set>
using namespace std;
namespace TestTest
{
class TestOne
{
private:
        int On;
public:
        TestOne() {};
        ~TestOne() {};
};
class ClassTest
{
private:
        int a;
        char b;
        const char* data_;
        class Tes;
public:
        //测试 STL 模板类
        void TestString(string str);
        void TestStringPoint(string* strP);
        void TestVector(vector<string> vec);
        void TestMap(map<string,string> Ma);
        void TestSet(set<int> Se);
        void TestStringArray(string strss[3]);
        ClassTest(int a) :a(a) {};
        ~ClassTest() {};
};
```

(下页继续)

(续上页)

```
}
```

6.4.2 参数捕获代码

```
被测试函数: void TestVector(vector<string> vec);
int TestVector6Times = -1;
void ParamCaptureClassTest::ParamCapture_TestVector6(vector<int, std::string> vec)
{
        TestVector6Times++;
        Json::Value Root;
        Json::Value TestVector6_Root;
        const char* JsonFilePath = "TestVector6.json";
        if (TestVector6Times != 0) {
        } else {
                Json::Reader _reader;
                std::ifstream _ifs(JsonFilePath);
                _reader.parse(_ifs, Root);
                TestVector6_Root = Root["TestVector6" + std::to_
↪string(TestVector6Times)];
        }

        /*It is the 1 parameter: vec    TestVector6
         *
         * Parameters of the prototype:vector<int, std::string> vec
         */

        /*vec*/
        Json::Value vec_Root;
        int size_vec = vec.size();
        for (auto t = 0; t < size_vec; t++) {
                vec_Root.append(vec.at(t));
        }

        TestVector6_Root.append(vec_Root);

        Root["TestVector6" + std::to_string(TestVector6Times)] = TestVector6_Root;
        std::ofstream JsonFile;
        JsonFile.open(JsonFilePath);
        Json::StyledWriter sw;
```

(下页继续)

(续上页)

```
        JsonFile << sw.write(Root);
        JsonFile.close();

}
void ParamCaptureClassTest::GlobalCapture_TestVector6()
{
        const char* JsonFilePath = "paramcapturevalue/ClassTest/TestVector6.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(JsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value TestVector6_Root = Root["TestVector6" + std::to_
↪string(TestVector6Times)];

        Root["TestVector6" + std::to_string(TestVector6Times)] = TestVector6_Root;
        std::ofstream JsonFile;
        Json::StyledWriter sw;
        JsonFile.open(JsonFilePath);
        JsonFile << sw.write(Root);
        JsonFile.close();

}
void ParamCaptureClassTest::ReturnCapture_TestVector6()
{
        const char* JsonFilePath = "paramcapturevalue/ClassTest/TestVector6.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(JsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value TestVector6_Root = Root["TestVector6" + std::to_
↪string(TestVector6Times)];

        Root["TestVector6" + std::to_string(TestVector6Times)] = TestVector6_Root;
        std::ofstream JsonFile;
        Json::StyledWriter sw;
        JsonFile.open(JsonFilePath);
        JsonFile << sw.write(Root);
        JsonFile.close();

}
```

(下页继续)

(续上页)

```
被测试函数: void TestMap(map<string,string> Ma);
int TestMap7Times = -1;
void ParamCaptureClassTest::ParamCapture_TestMap7(map<std::string, std::string> Ma)
{
        TestMap7Times++;
        Json::Value Root;
        Json::Value TestMap7_Root;
        const char* JsonFilePath = "TestMap7.json";
        if (TestMap7Times != 0) {
        } else {
                Json::Reader _reader;
                std::ifstream _ifs(JsonFilePath);
                _reader.parse(_ifs, Root);
                TestMap7_Root = Root["TestMap7" + std::to_string(TestMap7Times)];
        }
        /*It is the 1 parameter: Ma    TestMap7
         *
         * Parameters of the prototype:map<std::string, std::string> Ma
         */
        /*Ma*/
        Json::Value Ma_Root;
        int size_Ma = Ma.size();
        for (auto i : Ma)
        {
                Ma_Root["Ma_0"] = i.first;
                Ma_Root["Ma_1"] = i.second;
        }
        TestMap7_Root.append(Ma_Root);
        Root["TestMap7" + std::to_string(TestMap7Times)] = TestMap7_Root;
        std::ofstream JsonFile;
        JsonFile.open(JsonFilePath);
        Json::StyledWriter sw;
        JsonFile << sw.write(Root);
        JsonFile.close();
}
void ParamCaptureClassTest::GlobalCapture_TestMap7()
{
        const char* JsonFilePath = "paramcapturevalue/ClassTest/TestMap7.json";
```

(下页继续)

(续上页)

```
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(JsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value TestMap7_Root = Root["TestMap7" + std::to_string(TestMap7Times)];
        Root["TestMap7" + std::to_string(TestMap7Times)] = TestMap7_Root;
        std::ofstream JsonFile;
        Json::StyledWriter sw;
        JsonFile.open(JsonFilePath);
        JsonFile << sw.write(Root);
        JsonFile.close();
}
void ParamCaptureClassTest::ReturnCapture_TestMap7()
{
        const char* JsonFilePath = "paramcapturevalue/ClassTest/TestMap7.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(JsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value TestMap7_Root = Root["TestMap7" + std::to_string(TestMap7Times)];
        Root["TestMap7" + std::to_string(TestMap7Times)] = TestMap7_Root;
        std::ofstream JsonFile;
        Json::StyledWriter sw;
        JsonFile.open(JsonFilePath);
        JsonFile << sw.write(Root);
        JsonFile.close();
}
```

6.4.3 参数捕获代码解析

对于每种 STL 容器的取值方式不同，所以大部分都需要特殊处理，如 map、pair、string 等（以下简介几种）；string 类型的处理：普通的直接作为字符串处理，string 数组则作为字符串数组；map 类型的处理：由于 map 最常用的就是一组 key 对应一组 value，所以采用只赋值这两组，对于它还存在的排列组合方式则不考虑；pair 类型的处理：pair 就是一组 map，所以相同的方式存放一组值就可以；vector 类型的处理：vector 通过遍历的方式，将所有值都取出来进行存放。

6.5 基本类型的参数捕获

6.5.1 测试 demo

```
#include"json/json.h"
#pragma warning(disable:4996)
#pragma once
#include <iostream>
#include <vector>
#include<map>
#include <set>
using namespace std;
namespace TestTest
{
        struct Test
        {
                int in;
                char che;
                char * chP;
        };
        enum Te
        {
                one,
                two
        };
        class TestOne
        {
        private:
                int On;
        public:
                TestOne() {};
                ~TestOne() {};
                //测试枚举类型
        };
        class ClassTest
        {
        private:
                int a;
                char b;
                const char* data_;
```

(下页继续)

(续上页)

```
                class Tes;
        public:
                //测试基本类型
                int TestIntReturn(int Param);
                char TestCharReturn(char ch);
                Test TeststructReturn(Test te);
                const char* TestCharPoint();
                TestOne TestClassReturn();
                void TestClassFun(TestOne tss);
                void TestFun1(double de, long lo);
                void TestFun2(short sh, float fl);
                void TestFun(uint16_t uin, unsigned char ch);
                void TestFun4(unsigned long lo);
                //测试类类型
                void TestClassbuitin(TestOne te);
                void TestClassPoint(TestOne* Poin);
                //测试基本类型的指针
                void TestIntPoint(int* a,char * ch);
                void TestdoublePonit(long* lo,double* d);
                ClassTest(int a) :a(a) {};
                ~ClassTest() {};
        };
}
```

6.5.2 参数捕获代码

```
被测试函数: void TestIntPoint(int* a,char * ch);
int TestIntPoint2Times = -1;
void ParamCaptureClassTest::ParamCapture_TestIntPoint2(int* a, char* ch)
{
        TestIntPoint2Times++;
        Json::Value Root;
        Json::Value TestIntPoint2_Root;
        const char* JsonFilePath = "paramcapturevalue/ClassTest/TestIntPoint2.json";
        if (TestIntPoint2Times != 0) {
        } else {
                Json::Reader _reader;
                std::ifstream _ifs(JsonFilePath);
                _reader.parse(_ifs, Root);
```

(下页继续)

(续上页)

```
                TestIntPoint2_Root = Root["TestIntPoint2" + std::to_
↪string(TestIntPoint2Times)];
        }

        /*It is the 1 parameter: a    TestIntPoint2
         *
         * Parameters of the prototype:int *a
         */

        /*a*/
        Json::Value Arr_a;
        for (int row = 0; row < 1; row++) {
                Arr_a.append(Json::Value(a[row]));
        }
        TestIntPoint2_Root["a"] = Arr_a;

        /*It is the 2 parameter: ch    TestIntPoint2
         *
         * Parameters of the prototype:char *ch
         */

        /*ch*/
        TestIntPoint2_Root["ch"] = Json::Value(ch);

        Root["TestIntPoint2" + std::to_string(TestIntPoint2Times)] = TestIntPoint2_Root;
        std::ofstream JsonFile;
        JsonFile.open(JsonFilePath);
        Json::StyledWriter sw;
        JsonFile << sw.write(Root);
        JsonFile.close();

}
被测试函数：void TestdoublePonit(long* lo,double* d);
int TestdoublePonit3Times = -1;
void ParamCaptureClassTest::ParamCapture_TestdoublePonit3(long* lo, double* d)
{
        TestdoublePonit3Times++;
        Json::Value Root;
        Json::Value TestdoublePonit3_Root;
        const char* JsonFilePath = "paramcapturevalue/ClassTest/TestdoublePonit3.json";
```

(下页继续)

(续上页)

```
        if (TestdoublePonit3Times != 0) {
        } else {
                Json::Reader _reader;
                std::ifstream _ifs(JsonFilePath);
                _reader.parse(_ifs, Root);
                TestdoublePonit3_Root = Root["TestdoublePonit3" + std::to_
↪string(TestdoublePonit3Times)];
        }

        /*It is the 1 parameter: lo    TestdoublePonit3
         *
         * Parameters of the prototype:long *lo
         */

        /*lo*/
        Json::Value Arr_lo;
        for (int row = 0; row < 1; row++) {
                Arr_lo.append(Json::Value(lo[row]));
        }
        TestdoublePonit3_Root["lo"] = Arr_lo;

        /*It is the 2 parameter: d    TestdoublePonit3
         *
         * Parameters of the prototype:double *d
         */

        /*d*/
        Json::Value Arr_d;
        for (int row = 0; row < 1; row++) {
                Arr_d.append(Json::Value(d[row]));
        }
        TestdoublePonit3_Root["d"] = Arr_d;

        Root["TestdoublePonit3" + std::to_string(TestdoublePonit3Times)] =␣
↪TestdoublePonit3_Root;
        std::ofstream JsonFile;
        JsonFile.open(JsonFilePath);
        Json::StyledWriter sw;
        JsonFile << sw.write(Root);
        JsonFile.close();
```

(下页继续)

(续上页)

```
}
```

6.6 类类型和结构体类型的参数捕获

6.6.1 测试 demo

```
#include"json/json.h"
#pragma warning(disable:4996)
#pragma once
#include <iostream>
#include <string>
class WingsClassCapture;
namespace mySpace
{
        struct student
        {
                int age_;
                std::string name_;
                bool sex;
                FILE *fptr;
        };
        class ClassSecond;
        class ClassFirst
        {
        public:
                friend WingsClassCapture;
                ClassFirst() {}
                ClassFirst(int i ,student s) :  num_(i), stu_(s)
                {
                }
        private:
                int num_;
                void *ptr_;
                int(*mma)(int, int);
                FILE *fptr;
                student stu_;
                ClassSecond *chd_;
        };
```

(下页继续)

(续上页)

```
        class ClassSecond
        {
        public:
                friend WingsClassCapture;
                ClassSecond() {}
                int num_;
                void *ptr_;
                int(*mma)(int, int);
                FILE *fptr;
                ClassFirst chd_;
        };
        class ClassTest
        {
        public:
                friend WingsClassCapture;
                //类中包含结构体
                ClassFirst showtestobj(ClassFirst obj);
                ClassFirst* showtestptr(ClassFirst *ptr);
                student showstructtestobj(student obj);
                student* showstructtestptr(student *ptr);
        };
}
```

6.6.2 参数捕获代码

```
被测试函数: ClassFirst showtestobj(ClassFirst obj);
int ParamCapturemySpace_ClassTestshowtestobj0Times = -1;
void ParamCapturemySpace_ClassTest::ParamCapture_showtestobj0(mySpace::ClassFirst obj)
{
        ParamCapturemySpace_ClassTestshowtestobj0Times++;
        Json::Value Root;
        Json::Value showtestobj0_Root;
        const char* JsonFilePath = "D:/showtestobj0.json";
        if (ParamCapturemySpace_ClassTestshowtestobj0Times != 0) {
        } else {
                Json::Reader _reader;
                std::ifstream _ifs(JsonFilePath);
                _reader.parse(_ifs, Root);
                showtestobj0_Root = Root["showtestobj0" + std::to_
↪string(ParamCapturemySpace_ClassTestshowtestobj0Times)];
```

(下页继续)

(续上页)

```
        }

        /*It is the 1 parameter: obj    showtestobj0
         *
         * Parameters of the prototype:mySpace::ClassFirst obj
         */

        /* obj */
        WingsClassCapture wings_capture_class;
        showtestobj0_Root["obj"] = wings_capture_class.mySpace_ClassFirst_W_
↪MemberVarCaputer(obj);

        Root["showtestobj0" + std::to_string(ParamCapturemySpace_
↪ClassTestshowtestobj0Times)] = showtestobj0_Root;
        std::ofstream JsonFile;
        JsonFile.open(JsonFilePath);
        Json::StyledWriter sw;
        JsonFile << sw.write(Root);
        JsonFile.close();
}
被测试函数: ClassFirst* showtestptr(ClassFirst *ptr);
int ParamCapturemySpace_ClassTestshowtestptr1Times = -1;
void ParamCapturemySpace_ClassTest::ParamCapture_showtestptr1(mySpace::ClassFirst* ptr)
{
        ParamCapturemySpace_ClassTestshowtestptr1Times++;
        Json::Value Root;
        Json::Value showtestptr1_Root;
        const char* JsonFilePath = "D:/showtestptr1.json";
        if (ParamCapturemySpace_ClassTestshowtestptr1Times != 0) {
        } else {
                Json::Reader _reader;
                std::ifstream _ifs(JsonFilePath);
                _reader.parse(_ifs, Root);
                showtestptr1_Root = Root["showtestptr1" + std::to_
↪string(ParamCapturemySpace_ClassTestshowtestptr1Times)];
        }
        /*It is the 1 parameter: ptr    showtestptr1
         *
         * Parameters of the prototype:mySpace::ClassFirst *ptr
         */
```

(下页继续)

(续上页)

```
        /* ptr */
        if (ptr != nullptr) {
                WingsClassCapture wings_capture_class;
                showtestptr1_Root["ptr"] = wings_capture_class.mySpace_ClassFirst_W_
↪MemberVarCaputer(*ptr);
        }
        Root["showtestptr1" + std::to_string(ParamCapturemySpace_
↪ClassTestshowtestptr1Times)] = showtestptr1_Root;
        std::ofstream JsonFile;
        JsonFile.open(JsonFilePath);
        Json::StyledWriter sw;
        JsonFile << sw.write(Root);
        JsonFile.close();
}
被测函数: student showstructtestobj(student obj);
int ParamCapturemySpace_ClassTestshowstructtestobj2Times = -1;
void ParamCapturemySpace_ClassTest::ParamCapture_showstructtestobj2(mySpace::student obj)
{
        ParamCapturemySpace_ClassTestshowstructtestobj2Times++;
        Json::Value Root;
        Json::Value showstructtestobj2_Root;
        const char* JsonFilePath = "paramcapturevalue/mySpace_ClassTest/
↪showstructtestobj2.json";
        if (ParamCapturemySpace_ClassTestshowstructtestobj2Times != 0) {
        } else {
                Json::Reader _reader;
                std::ifstream _ifs(JsonFilePath);
                _reader.parse(_ifs, Root);
                showstructtestobj2_Root = Root["showstructtestobj2" + std::to_
↪string(ParamCapturemySpace_ClassTestshowstructtestobj2Times)];
        }

        /*It is the 1 parameter: obj    showstructtestobj2
         *
         * Parameters of the prototype:mySpace::student obj
         */

        /* obj */
        Json::Value obj_Root;
```

(下页继续)

(续上页)

```
        obj_Root = Paramstruct_mySpace_student(obj_Root, obj);
        showstructtestobj2_Root["obj"] = obj_Root;

        Root["showstructtestobj2" + std::to_string(ParamCapturemySpace_
↪ClassTestshowstructtestobj2Times)] = showstructtestobj2_Root;
        std::ofstream JsonFile;
        JsonFile.open(JsonFilePath);
        Json::StyledWriter sw;
        JsonFile << sw.write(Root);
        JsonFile.close();
}
void ParamCapturemySpace_ClassTest::GlobalCapture_showstructtestobj2()
{
        const char* JsonFilePath = "paramcapturevalue/mySpace_ClassTest/
↪showstructtestobj2.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(JsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value showstructtestobj2_Root = Root["showstructtestobj2" + std::to_
↪string(ParamCapturemySpace_ClassTestshowstructtestobj2Times)];

        Root["showstructtestobj2" + std::to_string(ParamCapturemySpace_
↪ClassTestshowstructtestobj2Times)] = showstructtestobj2_Root;
        std::ofstream JsonFile;
        Json::StyledWriter sw;
        JsonFile.open(JsonFilePath);
        JsonFile << sw.write(Root);
        JsonFile.close();
}
void ParamCapturemySpace_ClassTest::ReturnCapture_showstructtestobj2(mySpace::student␣
↪returnType)
{
        const char* JsonFilePath = "paramcapturevalue/mySpace_ClassTest/
↪showstructtestobj2.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(JsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value showstructtestobj2_Root = Root["showstructtestobj2" + std::to_
↪string(ParamCapturemySpace_ClassTestshowstructtestobj2Times)];
```

(下页继续)

(续上页)

```
        /* returnType */
        Json::Value returnType_Root;
        returnType_Root = Paramstruct_mySpace_student(returnType_Root, returnType);
        showstructtestobj2_Root["returnType"] = returnType_Root;

        Root["showstructtestobj2" + std::to_string(ParamCapturemySpace_
↪ClassTestshowstructtestobj2Times)] = showstructtestobj2_Root;
        std::ofstream JsonFile;
        Json::StyledWriter sw;
        JsonFile.open(JsonFilePath);
        JsonFile << sw.write(Root);
        JsonFile.close();
}
```

生成的类的参数捕获调用函数

```
#include "WingsClassCapture.h"
#include "paramcapture_structorunion.h"

Json::Value WingsClassCapture::mySpace_ClassFirst_W_MemberVarCaputer(mySpace::ClassFirst␣
↪temp_wings)
{
        Json::Value mySpace_ClassFirst_Root;
        /*num_*/
        mySpace_ClassFirst_Root["num_"] = Json::Value(temp_wings.num_);
        /*ptr_*/
        mySpace_ClassFirst_Root["ptr_"] = Json::Value();
        /* stu_ */
        Json::Value stu__Root;
        stu__Root = Paramstruct_mySpace_student(stu__Root, temp_wings.stu_);
        mySpace_ClassFirst_Root["stu_"] = stu__Root;

        /* chd_ */
        if (temp_wings.chd_ != nullptr) {
                WingsClassCapture wings_capture_class;
                mySpace_ClassFirst_Root["chd_"] = wings_capture_class.mySpace_
↪ClassSecond_W_MemberVarCaputer(*temp_wings.chd_);
        }
        return mySpace_ClassFirst_Root;
}
```

(下页继续)

(续上页)

```
Json::Value WingsClassCapture::mySpace_ClassSecond_W_
↪MemberVarCaputer(mySpace::ClassSecond temp_wings)
{
        Json::Value mySpace_ClassSecond_Root;
        /*num_*/
        mySpace_ClassSecond_Root["num_"] = Json::Value(temp_wings.num_);
        /*ptr_*/
        mySpace_ClassSecond_Root["ptr_"] = Json::Value();
        /* chd_ */
        WingsClassCapture wings_capture_class;
        mySpace_ClassSecond_Root["chd_"] = wings_capture_class.mySpace_ClassFirst_W_
↪MemberVarCaputer(temp_wings.chd_);
        return mySpace_ClassSecond_Root;
}

Json::Value WingsClassCapture::mySpace_ClassTest_W_MemberVarCaputer(mySpace::ClassTest␣
↪temp_wings)
{
        Json::Value mySpace_ClassTest_Root;
        return mySpace_ClassTest_Root;
}
```

生成的结构体的参数捕获调用函数

```
#include "paramcapture_structorunion.h"
Json::Value Paramstruct_TestTest_Test(Json::Value Test_Root, struct TestTest::Test a)
{
        /*in*/
        Test_Root["in"] = Json::Value(a.in);
        /*che*/
        char _che[2];
        _che[0] = a.che;
        _che[1] = '\0';
        Test_Root["che"] = Json::Value(_che);
        /*chP*/
        Test_Root["chP"] = Json::Value(a.chP);
        return Test_Root;

}

Json::Value Paramstruct_TestTest_Test_Point(Json::Value ArrayRoot, struct␣
↪TestTest::Test* a, int row)
```

(下页继续)

(续上页)

```
{
        if (a == nullptr) {
                return ArrayRoot;
        }
        for (int i = 0; i < row; i++) {
                Json::Value Test_Root;
                /*in*/
                Test_Root["in"] = Json::Value(a[i].in);
                /*che*/
                char _che[2];
                _che[0] = a[i].che;
                _che[1] = '\0';
                Test_Root["che"] = Json::Value(_che);
                /*chP*/
                Test_Root["chP"] = Json::Value(a[i].chP);

                ArrayRoot.append(Test_Root);
        }
        return ArrayRoot;
}

Json::Value Paramstruct_TestTest_Test_PointPoint(Json::Value SECArrayRoot, struct␣
↪TestTest::Test** a, int row, int column)
{
        if (a == nullptr) {
                return SECArrayRoot;
        }
        for (int i = 0; i < row; i++) {
                Json::Value ArrayRoot;
                for (int j = 0; j < column; j++) {
                        Json::Value Test_Root;
                        /*in*/
                        Test_Root["in"] = Json::Value(a[i][j].in);

                        /*che*/
                        char _che[2];
                        _che[0] = a[i][j].che;
                        _che[1] = '\0';
                        Test_Root["che"] = Json::Value(_che);
```

(下页继续)

(续上页)

```
                        /*chP*/
                        Test_Root["chP"] = Json::Value(a[i][j].chP);

                        ArrayRoot.append(Test_Root);
                }
                SECArrayRoot.append(ArrayRoot);
        }
        return SECArrayRoot;
}
```

6.6.3 参数捕获代码解析

由上面代码可以看到，类、结构体都是通过生成对应的函数，然后调用该函数进行赋值操作，当在结构体、类的成员变量中含有类或结构体时，会重复这个过程，即调用相对应的类或结构体的参数捕获函数。

CHAPTER 7

类 A 包含类 B 指针，类 B 包含类 A 变量

7.1 源码（插装过后）

```
#include"json/json.h"
#pragma once

#include <iostream>
#include <string>

class MyPointer;

class MyObject
{
public:
        void show() const
        {
                std::cout << "this is MyObject" << std::endl;
                std::cout << "my data: " << data_ << std::endl;
                if (m_pointer_ != nullptr) std::cout << "m_object show(): " << std::endl;
        }
private:
        const std::string data_;
```

(下页继续)

(续上页)

```
        MyPointer* m_pointer_;
public:
friend class DriverMyObject;
MyObject(std::string data_, MyPointer *m_pointer_, bool Wings):data_(data_), m_pointer_
↪(m_pointer_)
{}
};

class MyPointer
{
public:
        void show() const
        {
                std::cout << "this is MyPointer" << std::endl;
                std::cout << "my data: " << data_ << std::endl;
                std::cout << "m_object show(): " << std::endl;
                m_object_.show();
        }
private:
        std::string data_;
        MyObject m_object_;
public:
friend class DriverMyPointer;
MyPointer(std::string data_, MyObject m_object_, bool Wings):data_(data_), m_object_(m_
↪object_)
{}
MyPointer()
{}
};
```

7.2 初版解决方法

驱动生成

1. 成员变量是类，将其类型存放在 vector 中（vector 中初始存放驱动生成所生成的类的类型）

2. 对该类进行驱动生成，处理其成员变量

3. 遇见成员变量为类，与 vector 中对比

4. 如果存在相同数据的停止，此成员变量指针赋值为空，普通的类对象不再处理

5. 不同将类型放入 vector 中，继续执行 2

7.2.1 出现问题

以 MyObject 类为例子

vector 中存放顺序：

MyObject，MyPointer，然后就遇到 MyPointer 中的成员变量 MyObject **m_object_**，与 MyObject 相同，所有停止赋值，不对它进行处理。

如果 MyObject 不存在默认构造函数，此时 m_object_ 就无法生成，出现错误

7.2.2 解决方法

在 3 中对比判断之前，先判断成员变量是否为指针，指针则对比，非指针，就存入 vector 中继续处理.

以 MyObject 类为例子

vector 中存放顺序：

1.MyObject

2.MyObject，MyPointer

3.MyObject，MyPointer，MyObject

4.MyPointer* **m_pointer_** ；为指针，且有相同数据。驱动生成 MyPointer* **m_pointer_** = nullptr；

7.3 驱动代码

主要是两个类的构造代码

```
//类 MyObject
DriverMyObject::DriverMyObject(Json::Value Root, int times)
{
        Json::Value _MyObject_Root = Root["MyObject" + std::to_string(times)];
        std::string _data_ = _MyObject_Root["data_"].asString();

        int _m_pointer_pointSize = 0;
        Json::Value _m_pointer_m_pointer__Root = _MyObject_Root["m_pointer_"][_m_pointer_
↪pointSize];
        std::string _m_pointer_data_ = _m_pointer_m_pointer__Root["data_"].asString();

        Json::Value _m_pointer_m_object_m_object__Root = _m_pointer_m_pointer__Root["m_
↪object_"];
```

(下页继续)

(续上页)

```
        std::string _m_pointer_m_object_data_ = _m_pointer_m_object_m_object__Root["data_
↪"].asString();

        MyPointer* _m_pointer_m_object_m_pointer_ = nullptr;

        MyObject _m_pointer_m_object_(_m_pointer_m_object_data_, _m_pointer_m_object_m_
↪pointer_, false);

        MyPointer* _m_pointer_ = new MyPointer(_m_pointer_data_, _m_pointer_m_object_,␣
↪false);

        _MyObject = new MyObject(_data_, _m_pointer_, false);
}
//类 MyPointer
DriverMyPointer::DriverMyPointer(Json::Value Root, int times)
{
        Json::Value _MyPointer_Root = Root["MyPointer" + std::to_string(times)];
        std::string _data_ = _MyPointer_Root["data_"].asString();

        Json::Value _m_object_m_object__Root = _MyPointer_Root["m_object_"];
        std::string _m_object_data_ = _m_object_m_object__Root["data_"].asString();

        MyPointer* _m_object_m_pointer_ = nullptr;

        MyObject _m_object_(_m_object_data_, _m_object_m_pointer_, false);

        _MyPointer = new MyPointer(_data_, _m_object_, false);
}
```

CHAPTER 8

模板容器类型

8.1 测试源码

测试类型：vector 的普通类型，指针类型，数组类型，模板参数为结构体指针，模板参数为 pair 类型（其他容器处理方法基本相同）

```
#pragma once

#include <iostream>
#include <string>
#include <vector>

using namespace std;

namespace MySpace
{
        class FileMateData
        {
        private:
                int matedata_;
                string mma;
        };
```

(下页继续)

(续上页)

```
        struct FileDate
        {

        };

        class STLTesting
        {
        public:

                std::vector<std::string> returnvector()
                {
                        return std::vector<std::string>();
                }
        private:
                std::string str_;
                std::string *pstr_;
                std::vector<std::string> builit_vec;
                std::vector<std::string> *pvec;
                std::vector<int> vec[2];
                std::vector<std::pair<int, FileDate>> *pvecs;
                std::vector<FileMateData *> *input;
        };

}
```

8.1.1 插装后的源码

因为需要对类中的成员变量进行赋值，所有会在驱动生成的时候对类进行一些插装，通过插装的驱动函数对类成员变量完成初始化赋值。插装构造函数额外添加一个 bool 变量防止出现构造函数冲突。

```
#include"json/json.h"
#pragma once

#include <iostream>
#include <string>
#include <vector>

using namespace std;
```

(下页继续)

(续上页)

```
namespace MySpace
{
        class FileMateData
        {
        private:
                int matedata_;
                string mma;
        public:
        FileMateData(int matedata_, std::string mma, bool Wings):matedata_(matedata_),␣
↪mma(mma)
        {}
};

        struct FileDate
        {

        };

        class STLTesting
        {
        private:
                std::string str_;
                std::string *pstr_;
                std::vector<std::string> builit_vec;
                std::vector<std::string> *pvec;
                std::vector<int> vec[2];
                std::vector<std::pair<int, FileDate>> *pvecs;
                std::vector<FileMateData *> *input;
        public:
        STLTesting(std::string str_, std::string *pstr_, std::vector<std::string> builit_
↪vec, std::vector<std::string> *pvec, std::vector<int> vec[2], std::vector<std::pair
↪<int, FileDate> > *pvecs, std::vector<FileMateData *> *input, bool Wings):str_(str_),␣
↪pstr_(pstr_), builit_vec(builit_vec), pvec(pvec), pvecs(pvecs), input(input)
        {/* vec */
                                for(unsigned int size = 0; size < 2; size++)
                                {
                                        this->vec[size] = vec[size];
                                }
                                }
        };
```

(下页继续)

(续上页)

```
}
```

8.2 驱动文件生成

驱动类的构造函数

头文件

```
#include "E:/wuqingwen/vscode/TestTemplate/TestTemplate/FileMateData.h"
#include "driver.h"
#include "driver_Enum.h"
#include "driver_structorunion.h"
class DriverSTLTesting {
public:
        DriverSTLTesting(Json::Value Root, int times);
        ~DriverSTLTesting();

private:
        MySpace::STLTesting* _STLTesting;
};
```

源文件

```
/*The total header file of the tested file needed */
/*The header file corresponding to the source file */
#include "driverSTLTesting.h"
DriverSTLTesting::DriverSTLTesting(Json::Value Root, int times)
{
        Json::Value _STLTesting_Root = Root["STLTesting" + std::to_string(times)];
        string _str_ = _STLTesting_Root["str_"].asString();

        std::string* _pstr_ = new std::string(_STLTesting_Root["pstr_"].asString());

        Json::Value _builit_vec_RootArr = _STLTesting_Root["builit_vec"];
        vector<std::string> _builit_vec;
        int _builit_vec_size = _builit_vec_RootArr.size();
        for (int builit_vec_row = 0; builit_vec_row < _builit_vec_size; builit_vec_
↪row++) {
                Json::Value _builit_vec_Root = _builit_vec_RootArr[builit_vec_row];
```

(下页继续)

(续上页)

```
                string _builit_vec_0 = _builit_vec_Root["builit_vec_0"].asString();

                _builit_vec.push_back(_builit_vec_0);
        }
        Json::Value _pvec_RootArr = _STLTesting_Root["pvec"];
        vector<std::string>* _pvec = new vector<std::string>();
        int _pvec_size = _pvec_RootArr.size();
        for (int pvec_row = 0; pvec_row < _pvec_size; pvec_row++) {
                Json::Value _pvec_Root = _pvec_RootArr[pvec_row];

                string _pvec_0 = _pvec_Root["pvec_0"].asString();

                _pvec->push_back(_pvec_0);
        }
        /* vec */
        vector<int> _vec[2];
        for (int len = 0; len < 2; len++) {
                Json::Value _vec_RootArr = _STLTesting_Root["vec"][len];

                int _vec_size = _vec_RootArr.size();
                for (int vec_row = 0; vec_row < _vec_size; vec_row++) {
                        Json::Value _vec_Root = _vec_RootArr[vec_row];

                        /* vec_0 */
                        int _vec_0 = _vec_Root["vec_0"].asInt();

                        _vec[len].push_back(_vec_0);
                }
        }
        Json::Value _pvecs_RootArr = _STLTesting_Root["pvecs"];
        vector<pair<int, MySpace::FileDate>>* _pvecs = new vector<pair<int,␣
↪MySpace::FileDate>>();
        int _pvecs_size = _pvecs_RootArr.size();
        for (int pvecs_row = 0; pvecs_row < _pvecs_size; pvecs_row++) {
                Json::Value _pvecs_Root = _pvecs_RootArr[pvecs_row];

                Json::Value _pvecs_0_RootArr = _pvecs_Root["pvecs_0"];
                Json::Value _pvecs_0_Root = _pvecs_0_RootArr[0];
                /* pvecs_0_0 */
```

(下页继续)

(续上页)

```
                int _pvecs_0_0 = _pvecs_0_Root["pvecs_0_0"].asInt();
                /* pvecs_0_1 */
                Json::Value _pvecs_0_1_Root = _pvecs_0_Root["pvecs_0_1"];

                struct MySpace::FileDate _pvecs_0_1 = DriverstructFileDate(_pvecs_0_1_
↪Root);

                std::pair<int, MySpace::FileDate> _pvecs_0;
                _pvecs_0 = std::pair<int, MySpace::FileDate>(_pvecs_0_0, _pvecs_0_1);

                _pvecs->push_back(_pvecs_0);
        }
        Json::Value _input_RootArr = _STLTesting_Root["input"];
        vector<MySpace::FileMateData*>* _input = new vector<MySpace::FileMateData*>();
        int _input_size = _input_RootArr.size();
        for (int input_row = 0; input_row < _input_size; input_row++) {
                Json::Value _input_Root = _input_RootArr[input_row];

                int _input_0pointSize = 0;
                Json::Value _input_0input_0_Root = _input_Root["input_0"][_input_
↪0pointSize];
                /* matedata_ */
                int _input_0matedata_ = _input_0input_0_Root["matedata_"].asInt();

                string _input_0mma = _input_0input_0_Root["mma"].asString();

                MySpace::FileMateData* _input_0 = new MySpace::FileMateData(_input_
↪0matedata_, _input_0mma, false);

                _input->push_back(_input_0);
        }
        _STLTesting = new MySpace::STLTesting(_str_, _pstr_, _builit_vec, _pvec, _vec, _
↪pvecs, _input, false);
}
```

8.2.1 string 类型赋值部分

直接从 json 中取出对应的值进行赋值.

```
std::string* _pstr_ = new std::string(_STLTesting_Root["pstr_"].asString());
```

8.2.2 vector 基础类型赋值

赋值过程：定义变量, json 取值，构造容器元素对象，填充进容器，循环执行。

```
Json::Value _builit_vec_RootArr = _STLTesting_Root["builit_vec"];
        vector<std::string> _builit_vec;
        int _builit_vec_size = _builit_vec_RootArr.size();
        for (int builit_vec_row = 0; builit_vec_row < _builit_vec_size; builit_vec_
↪row++) {
                Json::Value _builit_vec_Root = _builit_vec_RootArr[builit_vec_row];

                string _builit_vec_0 = _builit_vec_Root["builit_vec_0"].asString();

                _builit_vec.push_back(_builit_vec_0);
        }
```

8.2.3 vector 指针类型

```
Json::Value _pvec_RootArr = _STLTesting_Root["pvec"];
        vector<std::string>* _pvec = new vector<std::string>();
        int _pvec_size = _pvec_RootArr.size();
        for (int pvec_row = 0; pvec_row < _pvec_size; pvec_row++) {
                Json::Value _pvec_Root = _pvec_RootArr[pvec_row];

                string _pvec_0 = _pvec_Root["pvec_0"].asString();

                _pvec->push_back(_pvec_0);
        }
```

8.2.4 vector 数组类型

```
vector<int> _vec[2];
        for (int len = 0; len < 2; len++) {
                Json::Value _vec_RootArr = _STLTesting_Root["vec"][len];

                int _vec_size = _vec_RootArr.size();
```

(下页继续)

(续上页)

```
                for (int vec_row = 0; vec_row < _vec_size; vec_row++) {
                        Json::Value _vec_Root = _vec_RootArr[vec_row];

                        /* vec_0 */
                        int _vec_0 = _vec_Root["vec_0"].asInt();

                        _vec[len].push_back(_vec_0);
                }
        }
```

8.2.5 vector 参数为模板类

```
Json::Value _input_RootArr = _STLTesting_Root["input"];
        vector<MySpace::FileMateData*>* _input = new vector<MySpace::FileMateData*>();
        int _input_size = _input_RootArr.size();
        for (int input_row = 0; input_row < _input_size; input_row++) {
                Json::Value _input_Root = _input_RootArr[input_row];

                int _input_0pointSize = 0;
                Json::Value _input_0input_0_Root = _input_Root["input_0"][_input_
↪0pointSize];
                /* matedata_ */
                int _input_0matedata_ = _input_0input_0_Root["matedata_"].asInt();

                string _input_0mma = _input_0input_0_Root["mma"].asString();

                MySpace::FileMateData* _input_0 = new MySpace::FileMateData(_input_
↪0matedata_, _input_0mma, false);

                _input->push_back(_input_0);
        }
```

8.2.6 构造被测类对象

```
_STLTesting = new MySpace::STLTesting(_str_, _pstr_, _builit_vec, _pvec, _vec, _pvecs, _
↪input, false);
```

8.3 该类的测试用例生成

此为被测类的值文件容器默认为生成三组值生成元素的值，容器数组根据数组数生成元素的值.

```
    "STLTesting0" : {
  "builit_vec" : [
     {
        "builit_vec_0" : "G7B"
     },
     {
        "builit_vec_0" : "YKi"
     },
     {
        "builit_vec_0" : "pym"
     }
  ],
  "input" : [
     {
        "input_0" : [
           {
              "matedata_" : 390,
              "mma" : "Qth"
           }
        ]
     },
     {
        "input_0" : [
           {
              "matedata_" : 2501,
              "mma" : "Pke"
           }
        ]
     },
     {
        "input_0" : [
           {
              "matedata_" : 6069,
              "mma" : "xiD"
           }
        ]
     }
```

(下页继续)

(续上页)


```
  ],
  "pstr_" : "Vdd",
  "pvec" : [
     {
        "pvec_0" : "jKJ"
     },
     {
        "pvec_0" : "zwW"
     },
     {
        "pvec_0" : "LGT"
     }
  ],
  "pvecs" : [
     {
        "pvecs_0" : [
           {
              "pvecs_0_0" : 3141,
              "pvecs_0_1" : null
           }
        ]
     },
     {
        "pvecs_0" : [
           {
              "pvecs_0_0" : 9514,
              "pvecs_0_1" : null
           }
        ]
     },
     {
        "pvecs_0" : [
           {
              "pvecs_0_0" : 348,
              "pvecs_0_1" : null
           }
        ]
     }
  ],
  "str_" : "5pD",
```


(下页继续)

(续上页)

```
   "vec" : [
      [
         {
            "vec_0" : 3805
         },
         {
            "vec_0" : 8407
         },
         {
            "vec_0" : 167
         }
      ],
      [
         {
            "vec_0" : 1839
         },
         {
            "vec_0" : 9528
         },
         {
            "vec_0" : 2960
         }
      ]
   ]
}
```

8.4 对参数有模板类的驱动测试

对之前的类添加成员函数进行测试, 在成员函数中对输入进去的数值进行打印。因为 fileDate 中没有类型, 这里我们只对了 pair 的 int 进行了打印。

```
void TestVectorPair(std::vector<std::pair<int, FileDate>> *pvecs);
```

8.4.1 驱动代码

```
int DriverSTLTestingTestVectorPair6(int times);
int TestVectorPair6Times;
/* Parameterized function processing,Root is the json for this file,Times is the number␣
↪of tests
```

(下页继续)

(续上页)

```
 * The function prototype:
* void TestVectorPair(std::vector<std::pair<int, FileDate> > *pvecs)
                        */
int DriverSTLTesting::DriverSTLTestingTestVectorPair6(int times)
{
        TestVectorPair6Times = times;
        /* Root is the json object of the value file.TestVectorPair6_Root is function.
↪TestVectorPair6 is json object.  */

        const char* jsonFilePath =
↪"E:\\wuqingwen\\vscode\\TemplateValueTest\\drivervalue/STLTesting/TestVectorPair6.json
↪";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value _TestVectorPair6_Root = Root["TestVectorPair6" + std::to_
↪string(times)];
        /*It is the 1 parameter: pvecs    TestVectorPair6
         *
         * Parameters of the prototype:std::vector<std::pair<int, FileDate> > *pvecs
         */

        Json::Value _pvecs_RootArr = _TestVectorPair6_Root["pvecs"];
        vector<std::pair<int, MySpace::FileDate>>* _pvecs = new vector<std::pair<int,␣
↪MySpace::FileDate>>();
        int _pvecs_size = _pvecs_RootArr.size();
        for (int pvecs_row = 0; pvecs_row < _pvecs_size; pvecs_row++) {
                Json::Value _pvecs_Root = _pvecs_RootArr[pvecs_row];

                Json::Value _pvecs_0_RootArr = _pvecs_Root["pvecs_0"];
                Json::Value _pvecs_0_Root = _pvecs_0_RootArr[0];
                /* pvecs_0_0 */
                int _pvecs_0_0 = _pvecs_0_Root["pvecs_0_0"].asInt();
                /* pvecs_0_1 */
                Json::Value _pvecs_0_1_Root = _pvecs_0_Root["pvecs_0_1"];

                struct MySpace::FileDate _pvecs_0_1 = DriverstructFileDate(_pvecs_0_1_
↪Root);
```

(下页继续)

(续上页)

```
                std::pair<int, struct MySpace::FileDate> _pvecs_0;
                _pvecs_0 = std::pair<int, struct MySpace::FileDate>(_pvecs_0_0, _pvecs_0_
↪1);

                _pvecs->push_back(_pvecs_0);
        }
        //The Function of Class    Call
        _STLTesting->TestVectorPair(_pvecs);

        return 0;
}
```

8.4.2 此函数值文件

```
{
   "TestVectorPair60" : {
          "pvecs" : [
                 {
                        "pvecs_0" : [
                           {
                                  "pvecs_0_0" : 1976,
                                  "pvecs_0_1" : null
                           }
                        ]
                 },
                 {
                        "pvecs_0" : [
                           {
                                  "pvecs_0_0" : 1829,
                                  "pvecs_0_1" : null
                           }
                        ]
                 },
                 {
                        "pvecs_0" : [
                           {
                                  "pvecs_0_0" : 3988,
                                  "pvecs_0_1" : null
                           }
```

(下页继续)

(续上页)

```
                    ]
                }
            ]
    }
}
```

8.5 支持类型

目前支持：vector，map，stack，set，list，deque，arrat

CHAPTER 9

对于 STL 标准库的 Gtest 部分的处理

9.1 测试例子及 Gtest 的 EXPECT_EQ 思想

以下例子用于测试 STL 标准库的内容，测试包含了 STL 标准库中的容器作为类的成员变量、结构体的成员变量、函数的返回值、函数的参数以及存在其一级指针和二级指针的情况，此外在考虑了容器之间嵌套和容器内置类型为类的情况。

```
#pragma once
#include <vector>
#include <set>
#include <map>
#include <list>
#include <iostream>
using namespace std;
class Testshare {
public:
        int a;
        char ch;
        string str;
        Testshare(int a) {
                cout << a << endl;
        };
        ~Testshare() {};
```

(下页继续)

(续上页)

```
};
struct TestStruct
{
        vector<int> TestV;
        vector<int> *TestVec;
        map<int, string> TestM;
        map<int, string> *TestMaC;
        set<string> *TestSetC;
        set<string> TestStrStr;
        string *str;
        string strt;
        vector<vector<string>> TestVecV;
        vector<map<int, string>> TestVVV;
        vector<map<int, string>> *TestVP;
        vector<vector<Testshare>> TestClass;
        vector<vector<Testshare>> *TestClP;
        map<int,vector<double>> TestMMM;
        map<int,vector<string, string>> *TestMP;
        unique_ptr <int> un1;
        pair<int, string> pa1;
        shared_ptr <string> p1;
        vector<Testshare> TestVecC;
        map<int,Testshare> TestMaC;
        set<Testshare> TestSetC;
};
class ClassTest
{
private:
        vector<int> TestV;
        vector<int> *TestVec;
        map<int, string> TestM;
        map<int, string> *TestMaC;
        set<string> *TestSetC;
        set<string> TestStrStr;
        string *str;
        string strt;
        vector<vector<string>> TestVecV;
        vector<map<int, string>> TestVVV;
        vector<map<int, string>> *TestVP;
        vector<vector<Testshare>> TestClass;
```

(下页继续)

(续上页)

```
        vector<vector<Testshare>> *TestClP;
        map<int, vector<double>> TestMMM;
        map<int, vector<string, string>> *TestMP;
        unique_ptr <int> un1;
        pair<int, string> pa1;
        shared_ptr <string> p1;
        vector<Testshare> TestVecC;
        map<int,Testshare> TestMaC;
        set<Testshare> TestSetC;
public:
        void TestvecReturn(vector<vector<string>> TestVec);
        void TestvecReturnP(vector<vector<string>> *TestVecP);
        void TestmapReturn(map<int, string> *Testmap);
        void Testsetreturn(set<map<int, string>> Testset);
        void TestsetreturnP(set<map<int,string>> *TestsetP);
        void TestCLReturn(set<Testshare> *Testsh);
        void TestStr(string *str);
        void testPairReturn(pair<int, string> *a);
        vector<Testshare> TestReturn();
        void TestStr(string str);
        vector<string> TestCaseVed();
        map<int, string> TestCaseMap();
        set<string> TestCaseSet();
        string TestStrReturn();
};
```

Gtest 的 EXPECT_EQ 思想

鉴于 STL 容器中存在内置类型，我们要确保两个容器中的内容完全一致（即相等），我们使用了将所有的容器进行展开进行比较，即比较两个容器内部进行展开比较。这种方法很好的解决了容器之间嵌套的问题，以及特殊类型作为容器的内置类型的情况（如类类型、string 等作为容器的内置类型）。

9.2 STL 标准库的容器作为类的成员变量

STL 标准库的容器作为类的成员变量，需要在我们的 GTest 文件中对类中的每个成员变量进行比较，对于 STL 容器需要对内部类型进行展开进行比较，分别从 _return_actual_Root、_expectreturn_expected_Root 中取出实际值和期望值，进行一层层取值比较。

```
void GtestExpectClassTest(Json::Value _return_actual_Root, Json::Value _expectreturn_
↪expected_Root)
{
        Json::Value _TestV_actual_Root = _return_actual_Root["TestV"];
        Json::Value _TestV_expected_Root = _expectreturn_expected_Root["TestV"];
        /* TestV */
        int size_TestV = _TestV_actual_Root.size();
        for (auto z = 0; z < size_TestV; z++) {
                /* TestV_0 */
                int _TestV_0_actual = _TestV_actual_Root["TestV_0"].asInt();
                /* TestV_0 */
                int _TestV_0_expected = _TestV_expected_Root["TestV_0"].asInt();
                /* TestV_0 */
                EXPECT_EQ(_TestV_0_actual, _TestV_0_expected);
        }
        Json::Value _TestVec_actual_Root = _return_actual_Root["TestVec"];
        Json::Value _TestVec_expected_Root = _expectreturn_expected_Root["TestVec"];
        /* TestVec */
        int size_TestVec = _TestVec_actual_Root.size();
        for (auto z = 0; z < size_TestVec; z++) {
                int size_TestVec_s = _TestVec_actual_Root[z].size();
                for (auto t = 0; t < size_TestVec_s; t++) {
                        /* TestVec_0 */
                        int _TestVec_0_actual = _TestVec_actual_Root["TestVec_0"].
↪asInt();
                        /* TestVec_0 */
                        int _TestVec_0_expected = _TestVec_expected_Root["TestVec_0"].
↪asInt();
                        /* TestVec_0 */
                        EXPECT_EQ(_TestVec_0_actual, _TestVec_0_expected);
                }
        }
        Json::Value _TestM_actual_Root = _return_actual_Root["TestM"];
        Json::Value _TestM_expected_Root = _expectreturn_expected_Root["TestM"];
        /* TestM */
        int size_TestM = _TestM_actual_Root.size();
        for (auto z = 0; z < size_TestM; z++) {
                /* TestM_0 */
                int _TestM_0_actual = _TestM_actual_Root["TestM_0"].asInt();
                /* TestM_0 */
```

(下页继续)

(续上页)

```
                int _TestM_0_expected = _TestM_expected_Root["TestM_0"].asInt();
                /* TestM_0 */
                EXPECT_EQ(_TestM_0_actual, _TestM_0_expected);
                Json::Value _TestM_1_actual_Root = _TestM_actual_Root["TestM_1"];
                Json::Value _TestM_1_expected_Root = _TestM_expected_Root["TestM_1"];
                string _TestM_1_actual = _TestM_1_actual_Root["TestM_1"].asString();
                string _TestM_1_expected = _TestM_1_expected_Root["TestM_1"].asString();
                EXPECT_EQ(_TestM_1_actual, _TestM_1_expected);
        }
        Json::Value _TestMaC_actual_Root = _return_actual_Root["TestMaC"];
        Json::Value _TestMaC_expected_Root = _expectreturn_expected_Root["TestMaC"];
        /* TestMaC */
        int size_TestMaC = _TestMaC_actual_Root.size();
        for (auto z = 0; z < size_TestMaC; z++) {
                int size_TestMaC_s = _TestMaC_actual_Root[z].size();
                for (auto t = 0; t < size_TestMaC_s; t++) {
                        /* TestMaC_0 */
                        int _TestMaC_0_actual = _TestMaC_actual_Root["TestMaC_0"].
↪asInt();
                        /* TestMaC_0 */
                        int _TestMaC_0_expected = _TestMaC_expected_Root["TestMaC_0"].
↪asInt();
                        /* TestMaC_0 */
                        EXPECT_EQ(_TestMaC_0_actual, _TestMaC_0_expected);
                        Json::Value _TestMaC_1_actual_Root = _TestMaC_actual_Root[
↪"TestMaC_1"];
                        Json::Value _TestMaC_1_expected_Root = _TestMaC_expected_Root[
↪"TestMaC_1"];
                        string _TestMaC_1_actual = _TestMaC_1_actual_Root["TestMaC_1"].
↪asString();
                        string _TestMaC_1_expected = _TestMaC_1_expected_Root["TestMaC_1
↪"].asString();
                        EXPECT_EQ(_TestMaC_1_actual, _TestMaC_1_expected);
                }
        }
}
```

9.3 StL 标准库的容器作为结构体的成员

结构体中包含容器的成员变量逻辑和类的成员变量是一致的，再次不过多进行赘述。

```
void GtestExpectTestStruct(Json::Value _return_actual_Root, Json::Value _expectreturn_
↪expected_Root){
        Json::Value _TestSetC_actual_Root = _return_actual_Root["TestSetC"];
        Json::Value _TestSetC_expected_Root = _expectreturn_expected_Root["TestSetC"];
        /* TestSetC */
        int size_TestSetC = _TestSetC_actual_Root.size();
        for (auto z = 0; z < size_TestSetC; z++) {
                int size_TestSetC_s = _TestSetC_actual_Root[z].size();
                for (auto t = 0; t < size_TestSetC_s; t++) {
                        string _TestSetC_0_actual = _return_actual_Root["TestSetC_0"].
↪asString();
                        string _TestSetC_0_expected = _expectreturn_expected_Root[
↪"TestSetC_0"].asString();
                        EXPECT_EQ(_TestSetC_0_actual, _TestSetC_0_expected);
                }
        }
        Json::Value _TestStrStr_actual_Root = _return_actual_Root["TestStrStr"];
        Json::Value _TestStrStr_expected_Root = _expectreturn_expected_Root["TestStrStr
↪"];
        /* TestStrStr */
        int size_TestStrStr = _TestStrStr_actual_Root.size();
        for (auto z = 0; z < size_TestStrStr; z++) {
                string _TestStrStr_0_actual = _return_actual_Root["TestStrStr_0"].
↪asString();
                string _TestStrStr_0_expected = _expectreturn_expected_Root["TestStrStr_0
↪"].asString();
                EXPECT_EQ(_TestStrStr_0_actual, _TestStrStr_0_expected);
        }
        string _str_actual = _return_actual_Root["str"].asString();
        string _str_expected = _expectreturn_expected_Root["str"].asString();
        EXPECT_EQ(_str_actual, _str_expected);
        string _strt_actual = _return_actual_Root["strt"].asString();
        string _strt_expected = _expectreturn_expected_Root["strt"].asString();
        EXPECT_EQ(_strt_actual, _strt_expected);
        Json::Value _TestVecV_actual_Root = _return_actual_Root["TestVecV"];
        Json::Value _TestVecV_expected_Root = _expectreturn_expected_Root["TestVecV"];
        /* TestVecV */
```

(下页继续)

(续上页)

```
        int size_TestVecV = _TestVecV_actual_Root.size();
        for (auto z = 0; z < size_TestVecV; z++) {
                /* TestVecV_0 */
                int size_TestVecV_0 = _TestVecV_actual_Root.size();
                for (auto z = 0; z < size_TestVecV_0; z++) {
                        string _TestVecV_0_0_actual = _return_actual_Root["TestVecV_0_0
↪"].asString();
                        string _TestVecV_0_0_expected = _expectreturn_expected_Root[
↪"TestVecV_0_0"].asString();
                        EXPECT_EQ(_TestVecV_0_0_actual, _TestVecV_0_0_expected);
                }
        }
        Json::Value _TestVVV_actual_Root = _return_actual_Root["TestVVV"];
        Json::Value _TestVVV_expected_Root = _expectreturn_expected_Root["TestVVV"];
        /* TestVVV */
        int size_TestVVV = _TestVVV_actual_Root.size();
        for (auto z = 0; z < size_TestVVV; z++) {
                /* TestVVV_0 */
                int size_TestVVV_0 = _TestVVV_actual_Root.size();
                for (auto z = 0; z < size_TestVVV_0; z++) {
                        /* TestVVV_0_0 */
                        int _TestVVV_0_0_actual = _TestVVV_0_actual_Root["TestVVV_0_0"].
↪asInt();
                        /* TestVVV_0_0 */
                        int _TestVVV_0_0_expected = _TestVVV_0_expected_Root["TestVVV_0_0
↪"].asInt();
                        /* TestVVV_0_0 */
                        EXPECT_EQ(_TestVVV_0_0_actual, _TestVVV_0_0_expected);
                        string _TestVVV_0_1_actual = _return_actual_Root["TestVVV_0_1"].
↪asString();
                        string _TestVVV_0_1_expected = _expectreturn_expected_Root[
↪"TestVVV_0_1"].asString();
                        EXPECT_EQ(_TestVVV_0_1_actual, _TestVVV_0_1_expected);
                }
        }
}
```

9.4 STL 作为函数返回值

当存在容器作为函数的返回值时，我们需要对函数的返回值进行 GTest 的比较，我们是从相应的 Json 的文件中取出对应的实际值和期望值，然后进行比较。

```
TEST_F(GtestClassTest, DriverClassTestTestcharReturnda7)
{
        const char* jsonFilePath = "drivervalue/ClassTest/TestcharReturnda7.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        for (int i = 0; i < CLASSTEST_TESTCHARRETURNDA7_TIMES; i++) {
                driverClassTest->DriverClassTestTestcharReturnda7(i);
                Json::Value _TestcharReturnda7_Root = Root["TestcharReturnda7" + std::to_
↪string(i)];
                /* return */
                std::string _return_actualStr = _TestcharReturnda7_actual_Root["return"].
↪asString();
                char _return_actual = _return_actualStr[0];
                /* return */
                std::string _return_expectedStr = _TestcharReturnda7_expected_Root[
↪"expectreturn"].asString();
                char _return_expected = _return_expectedStr[0];
                /* return_expected */
                EXPECT_EQ(_return_expected, _return_actual);
        }
}
TEST_F(GtestClassTest, DriverClassTestTestEnumCase8)
{
        const char* jsonFilePath = "drivervalue/ClassTest/TestEnumCase8.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        for (int i = 0; i < CLASSTEST_TESTENUMCASE8_TIMES; i++) {
                driverClassTest->DriverClassTestTestEnumCase8(i);
                Json::Value _TestEnumCase8_Root = Root["TestEnumCase8" + std::to_
↪string(i)];
                /* return */
                int _return_actual = _TestEnumCase8_actual_Root["return"].asInt();
```

(下页继续)

(续上页)

```
                /* return */
                int _return_expected = _TestEnumCase8_expected_Root["expectreturn"].
↪asInt();
                /* return */
                EXPECT_EQ(_return_actual, _return_expected);
        }
}

TEST_F(GtestClassTest, DriverClassTestTestIntretrun9)
{
        const char* jsonFilePath = "drivervalue/ClassTest/TestIntretrun9.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        for (int i = 0; i < CLASSTEST_TESTINTRETRUN9_TIMES; i++) {
                driverClassTest->DriverClassTestTestIntretrun9(i);
                Json::Value _TestIntretrun9_Root = Root["TestIntretrun9" + std::to_
↪string(i)];
                /* return */
                int _return_actual = _TestIntretrun9_actual_Root["return"].asInt();
                /* return */
                int _return_expected = _TestIntretrun9_expected_Root["expectreturn"].
↪asInt();
                /* return */
                EXPECT_EQ(_return_actual, _return_expected);
        }
}
```

9.5 STL 作为函数参数

```
int DriverClassTest::DriverClassTestTestCLReturn5(int times)
{
        TestCLReturn5Times = times;
        /* Root is the json object of the value file.TestCLReturn5_Root is function.
↪TestCLReturn5 is json object.  */

        const char* jsonFilePath = "drivervalue/ClassTest/TestCLReturn5.json";
```

(下页继续)

(续上页)

```
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value _TestCLReturn5_Root = Root["TestCLReturn5" + std::to_string(times)];
        /*It is the 1 parameter: Testsh    TestCLReturn5
         *
         * Parameters of the prototype:set<Testshare> *Testsh
         */

        Json::Value _Testsh_RootArr = _TestCLReturn5_Root["Testsh"];
        set<Testshare>* _Testsh = new set<Testshare>();
        int _Testsh_size = _Testsh_RootArr.size();
        for (int Testsh_row = 0; Testsh_row < _Testsh_size; Testsh_row++) {
                Json::Value _Testsh_Root = _Testsh_RootArr[Testsh_row];

                Json::Value _Testsh_0Testsh_0_Root = _Testsh_Root["Testsh_0"];
                /* a */
                int _Testsh_0a = _Testsh_0Testsh_0_Root["a"].asInt();

                /* ch */
                std::string _Testsh_0chStr = _Testsh_0Testsh_0_Root["ch"].asString();
                char _Testsh_0ch = _Testsh_0chStr[0];

                string _Testsh_0str = _Testsh_0Testsh_0_Root["str"].asString();

                Testshare _Testsh_0(_Testsh_0a, _Testsh_0ch, _Testsh_0str, false);
        }
        //The Function of Class    Call
        _ClassTest->TestCLReturn(_Testsh);
        return 0;
}
/* Parameterized function processing,Root is the json for this file,Times is the number␣
↪of tests

 * The function prototype:

* void TestStr(std::string *str)
          */
  int DriverClassTest::DriverClassTestTestStr6(int times)
```

(下页继续)

(续上页)

```
 {
 TestStr6Times = times;
 /* Root is the json object of the value file.TestStr6_Root is function.TestStr6 is␣
↪json object.  */

 const char* jsonFilePath = "drivervalue/ClassTest/TestStr6.json";
 Json::Value Root;
 Json::Reader _reader;
 std::ifstream _ifs(jsonFilePath);
 _reader.parse(_ifs, Root);
 Json::Value _TestStr6_Root = Root["TestStr6" + std::to_string(times)];
 /*It is the 1 parameter: str    TestStr6
  *

  * Parameters of the prototype:std::string *str
        */

 std::string* _str = new std::string(_TestStr6_Root["str"].asString());

 //The Function of Class    Call
 _ClassTest->TestStr(_str);

 return 0;
 }
```

9.6 处理 STL 的一级指针、二级指针

对于 STL 容器中的一级指针我们视其为一维数组，进行数组的展开比较，在每一层的数组中进行容器的展开比较，即将容器的内置类型进行展开，与上面的逻辑是相同的。

```
void GtestExpectClassTestPoint(Json::Value ClassTestreturn_actual_Root, Json::Value␣
↪ClassTestexpectreturn_expected_Root)
{
        int W_index_return = ClassTestreturn_actual_Root.size();
        int W_index_expectreturn = ClassTestexpectreturn_expected_Root.size();
        if (W_index_return == W_index_expectreturn && W_index_expectreturn != 0) {
                for (int i = 0; i < W_index_expectreturn; i++) {
                        Json::Value _return_actual_Root = ClassTestreturn_actual_Root[i];
                        Json::Value _expectreturn_expected_Root = ClassTestexpectreturn_
↪expected_Root[i];
```

(下页继续)

(续上页)

```
                        Json::Value _TestV_actual_Root = _return_actual_Root["TestV"];
                        Json::Value _TestV_expected_Root = _expectreturn_expected_Root[
↪"TestV"];
                        /* TestV */
                        int size_TestV = _TestV_actual_Root.size();
                        for (auto z = 0; z < size_TestV; z++) {

                                /* TestV_0 */
                                int _TestV_0_actual = _TestV_actual_Root["TestV_0"].
↪asInt();
                                /* TestV_0 */
                                int _TestV_0_expected = _TestV_expected_Root["TestV_0"].
↪asInt();
                                /* TestV_0 */
                                EXPECT_EQ(_TestV_0_actual, _TestV_0_expected);
                        }

                        Json::Value _TestVec_actual_Root = _return_actual_Root["TestVec
↪"];
                        Json::Value _TestVec_expected_Root = _expectreturn_expected_Root[
↪"TestVec"];
                        /* TestVec */
                        int size_TestVec = _TestVec_actual_Root.size();
                        for (auto z = 0; z < size_TestVec; z++) {
                                int size_TestVec_s = _TestVec_actual_Root[z].size();
                                for (auto t = 0; t < size_TestVec_s; t++) {

                                        /* TestVec_0 */
                                        int _TestVec_0_actual = _TestVec_actual_Root[
↪"TestVec_0"].asInt();
                                        /* TestVec_0 */
                                        int _TestVec_0_expected = _TestVec_expected_Root[
↪"TestVec_0"].asInt();
                                        /* TestVec_0 */
                                        EXPECT_EQ(_TestVec_0_actual, _TestVec_0_
↪expected);
                                }
                        }

                        Json::Value _TestM_actual_Root = _return_actual_Root["TestM"];
```

(下页继续)

(续上页)

```
                        Json::Value _TestM_expected_Root = _expectreturn_expected_Root[
↪"TestM"];
                        /* TestM */
                        int size_TestM = _TestM_actual_Root.size();
                        for (auto z = 0; z < size_TestM; z++) {

                                /* TestM_0 */
                                int _TestM_0_actual = _TestM_actual_Root["TestM_0"].
↪asInt();
                                /* TestM_0 */
                                int _TestM_0_expected = _TestM_expected_Root["TestM_0"].
↪asInt();
                                /* TestM_0 */
                                EXPECT_EQ(_TestM_0_actual, _TestM_0_expected);

                                Json::Value _TestM_1_actual_Root = _TestM_actual_Root[
↪"TestM_1"];
                                Json::Value _TestM_1_expected_Root = _TestM_expected_
↪Root["TestM_1"];

                                string _TestM_1_actual = _TestM_1_actual_Root["TestM_1"].
↪asString();
                                string _TestM_1_expected = _TestM_1_expected_Root["TestM_
↪1"].asString();
                                EXPECT_EQ(_TestM_1_actual, _TestM_1_expected);
                        }
}
```

9.7 特殊情况

当一些特殊类型作为容器的内置类型时，我们有不同的处理方式（暂时存在 string、类类型后续会支持更多）。

string

string 类型本身是容器，但它的比较是不需要进行展开的，我们是直接进行转化为 string 类型，即直接进行字符串比较。

class（类类型）

两个类之间是无法直接比较的，我们的处理方法是对类进行展开，比较类中的成员变量的值是否相同，对于类我们生成相应的成员变量比较的函数，当容器中出现类类型时，调用相应的函数比较函数即可。

CHAPTER 10

自定义模板类

思想：自定义模板类本质上还是类，只是它没有固定的类型，只有在模板类实例化时，才能知道具体的类型；我们的解决方案是将模板类的信息存储好，标记为模板类，在实际调用到该模板类类型时，如模板类作为参数，对该模板类的中的信息替换为当前传入的信息，即主要替换模板类的参数类型信息，生成一个新的模板类，其实这里已经是一个新的普通类了，没有了模板类的类型限制，通过走类的逻辑进行参数赋值，但最后在进行模板类的实例化时和类的有所不同，这里是对其进行判断之后进行模板类的类型实例化。实际遇到模板类时，根据实际的参数通过在缓存中替换数据构造一个新的“模板类”。

10.1 测试的源码

源码测试考虑到了模板类型的普通成员变量、一级指针、二级指针；以及模板类型作为函数参数和模板类型作为模板类型的内部类型（即模板类型的嵌套），对于复杂的模板类暂未处理，后续会持续增加。

```
#pragma warning(disable:4996)
#pragma once
#include <iostream>
#include <string>
#include <fstream>
#include <vector>
#include "json/json.h"
template <typename T>
class NestingTem
{
```

(下页继续)

(续上页)

```
public:
        NestingTem(T t,bool wings) :Test(t)
        {};
        ~NestingTem()
        {};
private:
        T Test;
};
// 类模版
template <typename T1,typename T2, typename T3, typename T4, typename T5>
class ClassTemplate
{
public:
        ClassTemplate() {};
        ClassTemplate(T1 t,T2 tt,T3 *t3,T4 t4[],T5 **t5,bool wings) :name(t),sex(tt),
↪Pointer(t3),TwoPinter(t5)
        {
                for (unsigned int size = 0; size < 10; size++)
                {
                        t4[size] = Arrayss[size];
                }
        }
private:
        T1 name;
        T2 sex;
        T3 *Pointer;
        T4 Arrayss[10];
        T5 **TwoPinter;
};

class Templatecl
{
public:
        Templatecl(ClassTemplate<int, double, char, int, int> _ts, ClassTemplate<int,␣
↪double, char, int, int> *_Pointer,
                ClassTemplate<int, double, char, int, int> **_TwoPointer,bool wings)␣
↪:ts(_ts),Pointer(_Pointer),TwoPointer(_TwoPointer){
        }
        void fprinTemplate(ClassTemplate<int, double,char,int,int> &m);
        void NestingTemFun1(ClassTemplate<NestingTem<int>, double, char, int, int > &
↪nesting);
```

(下页继续)

(续上页)

```
        void complexTemplate(ClassTemplate <NestingTem<char>,TemplateTwo<NestingTem<int>,
↪double>,char, int, int> &test2);
private:
        ClassTemplate<int, double,char,int,int> ts;
        ClassTemplate<int, double, char, int, int> *Pointer;
        ClassTemplate<int, double, char, int, int> **TwoPointer;
};
```

10.2 自定义模板类类型的成员变量

自定义模板类的成员变量在驱动中主要是在构造函数中初始化，即赋值，通过对模板类中的类型进行替换成实际的类型。其中包含普通类型、一级指针、二级指针。

```
普通类型、一级指针、二级指针的构造函数的赋值
DriverTemplatecl::DriverTemplatecl(Json::Value Root,int times)
 {
        Json::Value _Templatecl_Root = Root["Templatecl"+std::to_string(times)];
        Json::Value _tsts_Root=_Templatecl_Root["ts"];
/* name */
        int _tsname = _tsts_Root["name"].asInt();
/* sex */
        double _tssex = _tsts_Root["sex"].asDouble();
/* Pointer */
        char* _tsPointer;
        {
                std::string _tsPointer_str = _tsts_Root["Pointer"].asString();
                _tsPointer=new char[_tsPointer_str.size()];
                memcpy(_tsPointer,_tsPointer_str.c_str(),_tsPointer_str.size());
        }
/* Arrayss */
        int _tsArrayss[10];
        for(int len = 0; len < 10; len++)
        {
                _tsArrayss[len] = _tsts_Root["Arrayss"][len].asInt();
        }
/* TwoPinter */
        int **_tsTwoPinter;
        {
                int W_x = _tsts_Root["TwoPinter"].size();
```

(下页继续)

(续上页)

```
                _tsTwoPinter = new int*[W_x];
                for(int i = 0; i < W_x; i++)
                {
                        int W_y = _tsts_Root["TwoPinter"][i].size();
                        _tsTwoPinter[i] = new int[W_y];
                        for(int j = 0; j < W_y; j++)
                        {
                                _tsTwoPinter[i][j] = _tsts_Root["TwoPinter"][i][j].
↪asInt();
                        }
                }
        }
ClassTemplate<int, double, char, int, int> _ts(_tsname, _tssex, _tsPointer, _tsArrayss, _
↪tsTwoPinter, false);
int _PointerpointSize=0;
Json::Value _PointerPointer_Root=_Templatecl_Root["Pointer"][_PointerpointSize];
/* name */
        int _Pointername = _PointerPointer_Root["name"].asInt();
/* sex */
        double _Pointersex = _PointerPointer_Root["sex"].asDouble();
/* Pointer */
        char* _PointerPointer;
        {
                std::string _PointerPointer_str = _PointerPointer_Root["Pointer"].
↪asString();
                _PointerPointer=new char[_PointerPointer_str.size()];
                memcpy(_PointerPointer,_PointerPointer_str.c_str(),_PointerPointer_str.
↪size());
        }
/* Arrayss */
        int _PointerArrayss[10];
        for(int len = 0; len < 10; len++)
        {
                _PointerArrayss[len] = _PointerPointer_Root["Arrayss"][len].asInt();
        }
/* TwoPinter */
        int **_PointerTwoPinter;
        {
                int W_x = _PointerPointer_Root["TwoPinter"].size();
                _PointerTwoPinter = new int*[W_x];
```

(下页继续)

(续上页)

```
                for(int i = 0; i < W_x; i++)
                {
                        int W_y = _PointerPointer_Root["TwoPinter"][i].size();
                        _PointerTwoPinter[i] = new int[W_y];
                        for(int j = 0; j < W_y; j++)
                        {
                                _PointerTwoPinter[i][j] = _PointerPointer_Root["TwoPinter
↪"][i][j].asInt();
                        }
                }
        }

        ClassTemplate<int, double, char, int, int> *_Pointer=new ClassTemplate<int,␣
↪double, char, int, int>(_Pointername,
                _Pointersex, _PointerPointer, _PointerArrayss, _PointerTwoPinter, false);
        Json::Value _TwoPointerTwoPointer_Root=_Templatecl_Root["TwoPointer"];
/* name */
        int _TwoPointername = _TwoPointerTwoPointer_Root["name"].asInt();
/* sex */
        double _TwoPointersex = _TwoPointerTwoPointer_Root["sex"].asDouble();
/* Pointer */
        char* _TwoPointerPointer;
        {
                std::string _TwoPointerPointer_str = _TwoPointerTwoPointer_Root["Pointer
↪"].asString();
                _TwoPointerPointer=new char[_TwoPointerPointer_str.size()];
                memcpy(_TwoPointerPointer,_TwoPointerPointer_str.c_str(),_
↪TwoPointerPointer_str.size());
        }
/* Arrayss */
        int _TwoPointerArrayss[10];
        for(int len = 0; len < 10; len++)
        {
                _TwoPointerArrayss[len] = _TwoPointerTwoPointer_Root["Arrayss"][len].
↪asInt();
        }
/* TwoPinter */
        int **_TwoPointerTwoPinter;
        {
                int W_x = _TwoPointerTwoPointer_Root["TwoPinter"].size();
```

(下页继续)

(续上页)

```
                _TwoPointerTwoPinter = new int*[W_x];
                for(int i = 0; i < W_x; i++)
                {
                        int W_y = _TwoPointerTwoPointer_Root["TwoPinter"][i].size();
                        _TwoPointerTwoPinter[i] = new int[W_y];
                        for(int j = 0; j < W_y; j++)
                        {
                                _TwoPointerTwoPinter[i][j] = _TwoPointerTwoPointer_Root[
↪"TwoPinter"][i][j].asInt();
                        }
                }
        }
ClassTemplate<int, double, char, int, int> *_TwoPointer_ClassTemplate=new ClassTemplate
↪<int, double, char, int, int>(_TwoPointername,_TwoPointersex, _TwoPointerPointer, _
↪TwoPointerArrayss, _TwoPointerTwoPinter, false);
        ClassTemplate<int, double, char, int, int> **_TwoPointer=&_TwoPointer_
↪ClassTemplate;
        _Templatecl=new Templatecl(_ts, _Pointer, _TwoPointer, false);
}
```

10.3 自定义模板类类型的函数参数

普通的自定义模板类类型作为函数参数进行赋值驱动函数。

```
测试函数:
void fprinTemplate(ClassTemplate<int, double,char,int,int> & m);
驱动代码:
int DriverTemplatecl:: DriverTemplateclfprinTemplate0(int times)
{
        fprinTemplate0Times = times;
        const char*jsonFilePath="drivervalue/Templatecl/fprinTemplate0.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value _fprinTemplate0_Root = Root["fprinTemplate0"+std::to_string(times)];
        Json::Value _mm_Root=_fprinTemplate0_Root["m"];
/* name */
        int _mname = _mm_Root["name"].asInt();
```

(下页继续)

(续上页)

```
/* sex */
        double _msex = _mm_Root["sex"].asDouble();
/* Pointer */
        char* _mPointer;
        {
                 std::string _mPointer_str = _mm_Root["Pointer"].asString();
                 _mPointer=new char[_mPointer_str.size()];
                 memcpy(_mPointer,_mPointer_str.c_str(),_mPointer_str.size());
        }
/* Arrayss */
        int _mArrayss[10];
        for(int len = 0; len < 10; len++)
        {
                 _mArrayss[len] = _mm_Root["Arrayss"][len].asInt();
        }
/* TwoPinter */
   int **_mTwoPinter;
   {
               int W_x = _mm_Root["TwoPinter"].size();
               _mTwoPinter = new int*[W_x];
               for(int i = 0; i < W_x; i++)
               {
                       int W_y = _mm_Root["TwoPinter"][i].size();
                       _mTwoPinter[i] = new int[W_y];
                       for(int j = 0; j < W_y; j++)
                       {
                               _mTwoPinter[i][j] = _mm_Root["TwoPinter"][i][j].asInt();
                       }
                }
   }
   ClassTemplate<int, double, char, int, int> _m(_mname, _msex, _mPointer, _mArrayss, _
↪mTwoPinter, false);
//The Function of Class    Call
   _Templatecl->fprinTemplate( _m);
  return 0;
}
```

10.4 模板类作为自定义模板类的参数类型

模板类嵌套模板类，即模板类中的参数类型是其他的模板类，形成嵌套，下列代码是测试嵌套了三层的模板类参数。

```
测试的函数:
void complexTemplate(ClassTemplate <NestingTem<char>,TemplateTwo<NestingTem<int>,double>,
↪char, int, int> &test2);
驱动代码:
int DriverTemplatecl:: DriverTemplateclcomplexTemplate2(int times)
{
        complexTemplate2Times = times;
        const char*jsonFilePath="drivervalue/Templatecl/complexTemplate2.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value _complexTemplate2_Root = Root["complexTemplate2"+std::to_
↪string(times)];
        Json::Value _test2test2_Root=_complexTemplate2_Root["test2"];
        Json::Value _test2test2_0test2_0_Root=_test2test2_Root["test2_0"];
/* Test */
        std::string _test2test2_0TestStr = _test2test2_0test2_0_Root["Test"].asString();
        char _test2test2_0Test=_test2test2_0TestStr[0];
        NestingTem<char> _test2test2_0(_test2test2_0Test, false);
        Json::Value _test2test2_1test2_1_Root=_test2test2_Root["test2_1"];
        Json::Value _test2test2_1test2_1_0test2_1_0_Root=_test2test2_1test2_1_Root[
↪"test2_1_0"];
/* Test */
        int _test2test2_1test2_1_0Test = _test2test2_1test2_1_0test2_1_0_Root["Test"].
↪asInt();
        NestingTem<int> _test2test2_1test2_1_0(_test2test2_1test2_1_0Test, false);
/* Test2 */
        double _test2test2_1Test2 = _test2test2_1test2_1_Root["Test2"].asDouble();
        TemplateTwo<NestingTem<int>, double> _test2test2_1(_test2test2_1test2_1_0, _
↪test2test2_1Test2, false);
/* Pointer */
        char* _test2Pointer;
        {
                std::string _test2Pointer_str = _test2test2_Root["Pointer"].asString();
                _test2Pointer=new char[_test2Pointer_str.size()];
```

(下页继续)

(续上页)

```
                memcpy(_test2Pointer,_test2Pointer_str.c_str(),_test2Pointer_str.size());
        }
/* Arrayss */
        int _test2Arrayss[10];
        for(int len = 0; len < 10; len++)
        {
                _test2Arrayss[len] = _test2test2_Root["Arrayss"][len].asInt();
        }
/* TwoPinter */
        int **_test2TwoPinter;
        {
                int W_x = _test2test2_Root["TwoPinter"].size();
                _test2TwoPinter = new int*[W_x];
                for(int i = 0; i < W_x; i++)
                {
                        int W_y = _test2test2_Root["TwoPinter"][i].size();
                        _test2TwoPinter[i] = new int[W_y];
                        for(int j = 0; j < W_y; j++)
                        {
                                _test2TwoPinter[i][j] = _test2test2_Root["TwoPinter
↪"][i][j].asInt();
                        }
                }
        }
ClassTemplate<NestingTem<char>, TemplateTwo<NestingTem<int>, double>, char, int, int> _
↪test2(_test2test2_0, _test2test2_1, _test2Pointer, _test2Arrayss, _test2TwoPinter,␣
↪false);
//The Function of Class    Call
   _Templatecl->complexTemplate( _test2);
   return 0;
}
```

CHAPTER 11

枚举类型

11.1 测试的源码

测试的源码中考虑到了普通枚举、强枚举、枚举指针、枚举引用的值生成、驱动生成以及参数赋值驱动生成，以及构造函数中对类的枚举类型的成员变量的赋值。

```
#include "json/json.h"
#pragma once
namespace wings_grammars_test {
        enum Code {
                kOk = 0,
                kNotFound = 1,
                kCorruption = 2,
                kNotSupported = 3,
                kInvalidArgument = 4,
                kIOError = 5
        };
        enum class EnumBase
        {
                kNoCompression = 0x0,
                kSnappyCompression = 0x1
        };
        class EnumClassTesting
```

(下页继续)

(续上页)

```
        {
        private:
                EnumBase enumBaseType;
                EnumBase &enumBaseTypeR;
                Code codeType;
                Code *codePointType;
        public:
                void CodeFunc(Code codeType);
                void EnumBaseFunc(EnumBase enumBaseType);
                void EnumBaseFuncR(EnumBase &enumBaseType);
                void CodeFuncPoint(Code *codeType);
        public:
                EnumClassTesting(wings_grammars_test::EnumBase enumBaseType,
                wings_grammars_test::EnumBase &enumBaseTypeR,
                wings_grammars_test::Code codeType, wings_grammars_test::Code␣
↪*codePointType,
                bool Wings) :enumBaseType(enumBaseType), enumBaseTypeR(enumBaseTypeR),␣
↪codeType(codeType), codePointType(codePointType)
                {}
        };
}
```

11.2 枚举的测试用例生成

鉴于枚举驱动部分需要传入字符串进行比较然后赋值，所以值生成部分需要获取所有枚举的具体值（字符串类型的）；

如（KOK 等，非 0、1、2）：

```
<wings_grammars_test::Code>
        <kOk value="0" />
        <kNotFound value="1" />
        <kCorruption value="2" />
        <kNotSupported value="3" />
        <kInvalidArgument value="4" />
        <kIOError value="5" />
</wings_grammars_test::Code>
```

问题： 为什么只能使用字符串进行赋值?

答： 因为 c++11 存在强枚举类型，如：enum class Enumeration{ };

此种枚举为类型安全的，枚举类型不能隐式地转换为整数，也无法与整数数值做比较，所以只能赋值枚举类型相应的字符串。

问题：一个枚举类型存在多个值，怎么确定是哪一个值?

答：枚举类型存在多个值，同时也意味着每个值都可以赋值给原函数，所以采用随机数的方式，随机生成该枚举类型中的任一值进行赋值。

问题：枚举指针、枚举引用、基本枚举的值分别是什么形式?

答：枚举指针、基本枚举都是一样的，随机生成一个该类型的枚举值（字符串形式），将该字符串存储到 json 中；枚举指针视为数组类型，存储多个该类型的枚举值（字符串形式）。

生成之后的结果：

```
{
   "EnumClassTesting0" : {
          "codePointType" : [ "kInvalidArgument", "kIOError", "kCorruption" ],
          "codeType" : "kNotSupported",
          "enumBaseType" : "kSnappyCompression",
          "enumBaseTypeR" : "kNoCompression"
   }
}
```

11.3 普通枚举

普通枚举直接进行字符串比较，然后返回枚举名即可。

驱动代码：

```
wings_grammars_test::EnumBase DriverenumEnumBase(string Enumvalue)
{
        wings_grammars_test::EnumBase _EnumBase_value;
        if (Enumvalue == "kNoCompression") {
                _EnumBase_value = wings_grammars_test::EnumBase::kNoCompression;
        }
        else if (Enumvalue == "kSnappyCompression") {
                _EnumBase_value = wings_grammars_test::EnumBase::kSnappyCompression;
        }
        return _EnumBase_value;
}
```

普通枚举类型赋值代码：

```
int DriverEnumClassTesting::DriverEnumClassTestingEnumBaseFunc0(int times)
{
        EnumBaseFunc0Times = times;
        /* Root is the json object of the value file.EnumBaseFunc0_Root is function.
↪EnumBaseFunc0 is json object.  */
        const char* jsonFilePath = "drivervalue/EnumClassTesting/EnumBaseFunc0.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value _EnumBaseFunc0_Root = Root["EnumBaseFunc0" + std::to_string(times)];
        /*It is the 1 parameter: enumBaseType    EnumBaseFunc0
         *
         * Parameters of the prototype:wings_grammars_test::EnumBase enumBaseType
         */
        /* enumBaseType */
        string _EnumBase_enumBaseType = _EnumBaseFunc0_Root["enumBaseType"].asString();
        wings_grammars_test::EnumBase _enumBaseType = DriverenumEnumBase(_EnumBase_
↪enumBaseType);
        //The Function of Class    Call
        _EnumClassTesting->EnumBaseFunc(_enumBaseType);
        return 0;
}
```

11.4 枚举指针

枚举指针当做数组处理，进行相应的参数赋值，和值生成部分相对应起来。

问题：枚举指针为什么当做数组处理?

答：因为大部分的指针都可以当成数组处理，而且普通枚举值本身可以理解为和 int 是一样的，所以枚举指针的赋值可以作为数组处理。

驱动代码：

```
wings_grammars_test::EnumBase* DriverenumEnumBasePoint(string Enumvalue)
{
        wings_grammars_test::EnumBase _EnumBase_value;
        if (Enumvalue == "kNoCompression") {
                _EnumBase_value = wings_grammars_test::EnumBase::kNoCompression;
        }
```

(下页继续)

(续上页)

```
        else if (Enumvalue == "kSnappyCompression") {
                _EnumBase_value = wings_grammars_test::EnumBase::kSnappyCompression;
        }
        wings_grammars_test::EnumBase* _EnumBasePointer = &_EnumBase_value;
        return _EnumBasePointer;
}
```

枚举指针赋值驱动代码：

```
int DriverEnumClassTesting::DriverEnumClassTestingCodeFuncPoint3(int times)
{
        CodeFuncPoint3Times = times;
        /* Root is the json object of the value file.CodeFuncPoint3_Root is function.
↪CodeFuncPoint3 is json object.  */
        const char* jsonFilePath = "drivervalue/EnumClassTesting/CodeFuncPoint3.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value _CodeFuncPoint3_Root = Root["CodeFuncPoint3" + std::to_
↪string(times)];
        /*It is the 1 parameter: codeType    CodeFuncPoint3
         *
         * Parameters of the prototype:wings_grammars_test::Code *codeType
         */
        /* codeType */
        wings_grammars_test::Code* _codeType;
        {
                int W_index = _CodeFuncPoint3_Root["codeType"].size();
                _codeType = new wings_grammars_test::Code[W_index];
                for (int i = 0; i < W_index; i++) {
                        string _EnumValuecodeType = _CodeFuncPoint3_Root["codeType"][i].
↪asString();
                        _codeType[i] = DriverenumCode(_EnumValuecodeType);
                }
        }
        //The Function of Class    Call
        _EnumClassTesting->CodeFuncPoint(_codeType);
        return 0;
}
```

11.5 枚举引用

枚举引用的逻辑和普通枚举是一样的，只是在枚举驱动部分返回的是一个枚举的指针。

驱动代码：

```
wings_grammars_test::EnumBase& DriverenumLvalueEnumBase(string Enumvalue)
{
        wings_grammars_test::EnumBase* _EnumBase_value = new wings_grammars_
↪test::EnumBase();
        if (Enumvalue == "kNoCompression") {
                *_EnumBase_value = wings_grammars_test::EnumBase::kNoCompression;
        }
        else if (Enumvalue == "kSnappyCompression") {
                *_EnumBase_value = wings_grammars_test::EnumBase::kSnappyCompression;
        }
        return *_EnumBase_value;
}
```

枚举引用赋值驱动代码：

```
int DriverEnumClassTesting::DriverEnumClassTestingEnumBaseFuncR1(int times)
{
        EnumBaseFuncR1Times = times;
        /* Root is the json object of the value file.EnumBaseFuncR1_Root is function.
↪EnumBaseFuncR1 is json object.  */
        const char* jsonFilePath = "drivervalue/EnumClassTesting/EnumBaseFuncR1.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value _EnumBaseFuncR1_Root = Root["EnumBaseFuncR1" + std::to_
↪string(times)];
        /*It is the 1 parameter: enumBaseType    EnumBaseFuncR1
         *
         * Parameters of the prototype:wings_grammars_test::EnumBase &enumBaseType
         */
        /* enumBaseType */
        string _EnumBase_enumBaseType = _EnumBaseFuncR1_Root["enumBaseType"].asString();
        wings_grammars_test::EnumBase& _enumBaseType = DriverenumLvalueEnumBase(_
↪EnumBase_enumBaseType);
        //The Function of Class    Call
```

(下页继续)

(续上页)

```
        _EnumClassTesting->EnumBaseFuncR(_enumBaseType);
        return 0;
}
```

11.6 构造函数中对类中枚举类型的成员变量的赋值

因为构造函数中对枚举类型的成员变量的赋值调用的逻辑就是函数中对枚举类型的赋值，所以区别不大。

```
DriverEnumClassTesting::DriverEnumClassTesting(Json::Value Root, int times)
{
        Json::Value _EnumClassTesting_Root = Root["EnumClassTesting" + std::to_
↪string(times)];
        /* enumBaseType */
        string _EnumBase_enumBaseType = _EnumClassTesting_Root["enumBaseType"].
↪asString();
        wings_grammars_test::EnumBase _enumBaseType = DriverenumEnumBase(_EnumBase_
↪enumBaseType);
        /* enumBaseTypeR */
        string _EnumBase_enumBaseTypeR = _EnumClassTesting_Root["enumBaseTypeR"].
↪asString();
        wings_grammars_test::EnumBase& _enumBaseTypeR = DriverenumLvalueEnumBase(_
↪EnumBase_enumBaseTypeR);
        /* codeType */
        string _Code_codeType = _EnumClassTesting_Root["codeType"].asString();
        wings_grammars_test::Code _codeType = DriverenumCode(_Code_codeType);
        /* codePointType */
        wings_grammars_test::Code* _codePointType;
        {
                int W_index = _EnumClassTesting_Root["codePointType"].size();
                _codePointType = new wings_grammars_test::Code[W_index];
                for (int i = 0; i < W_index; i++) {
                        string _EnumValuecodePointType = _EnumClassTesting_Root[
↪"codePointType"][i].asString();
                        _codePointType[i] = DriverenumCode(_EnumValuecodePointType);
                }
        }
        _EnumClassTesting = new wings_grammars_test::EnumClassTesting(_enumBaseType, _
↪enumBaseTypeR, _codeType, _codePointType, false);
}
```

CHAPTER 12

结构体引用

12.1 测试的源码

测试的源码中考虑到了普通结构体、结构体指针、结构体引用作为参数，以及作为返回值，并且考虑了结构体中含有结构体成员的情况，构造函数中对类的结构体的引用类型的成员变量的赋值。

```
#include"json/json.h"
#include <iostream>
using namespace std;
#pragma once
class StructQuote
{
public:
        StructQuote()
        {
        };
        ~StructQuote() {};
        struct TestCase
        {
                int had;
        };
        struct QuoteTestCase
        {
```

(下页继续)

(续上页)

```
                int one;
                char two;
                TestCase CaseQuote;
        };
public:
        QuoteTestCase *CheckFunc;
        QuoteTestCase CheckFuncR;
        QuoteTestCase &CheckFuncRRR=CheckFuncR;
        void CheckNameTest(QuoteTestCase *CheckFunc);
        void CheckFunctionName(QuoteTestCase CheckFunc);
        void CheckFunctionTest(QuoteTestCase &CheckFunc);
        StructQuote::QuoteTestCase *CheckPointer(QuoteTestCase *CheckFunc);
        StructQuote::QuoteTestCase Checkbuiltn(QuoteTestCase CheckFunc);
        StructQuote::QuoteTestCase &CheckNameXX(QuoteTestCase CheckFunc);
public:
StructQuote(QuoteTestCase *CheckFunc, QuoteTestCase CheckFuncR, QuoteTestCase &
↪CheckFuncRRR , bool Wings):CheckFunc(CheckFunc), CheckFuncR(CheckFuncR),␣
↪CheckFuncRRR(CheckFuncRRR)
{}
};
```

12.2 结构体引用的值生成

结构体引用的值生成和结构体普通类型是一致的，直接对结构体内部的成员进行一个个赋值。

```
{
   "QuoteTestCase0" : {
          "CaseQuote" : {
                 "had" : 8039
          },
          "one" : 143,
          "two" : "s"
   },
   "StructQuote0" : {
          "CheckFunc" : [
                 {
                        "CaseQuote" : {
                           "had" : 6547
                        },
```

(下页继续)

(续上页)

```
                        "one" : 2732,
                        "two" : "v"
                }
        ],
        "CheckFuncR" : {
                "CaseQuote" : {
                        "had" : 5702
                },
                "one" : 2401,
                "two" : "Q"
        },
        "CheckFuncRRR" : {
                "CaseQuote" : {
                        "had" : 1486
                },
                "one" : 7740,
                "two" : "u"
        }
   },
   "TestCase0" : {
        "had" : 1764
   }
}
```

12.3 结构体引用作为参数

结构体引用直接调用普通结构体的赋值函数。

问题： 为什么直接调用普通结构体的赋值函数?

答： 因为结构体引用部分的逻辑和普通结构体是一模一样的，值生成部分也是一样，所以选择在参数赋值时直接调用普通结构体类型的，这样可以不用再写一遍引用部分的驱动。

```
int DriverStructQuote::DriverStructQuoteCheckFunctionTest2(int times)
{
        CheckFunctionTest2Times = times;
        /* Root is the json object of the value file.CheckFunctionTest2_Root is function.
↪CheckFunctionTest2 is json object.  */
        const char* jsonFilePath = "drivervalue/StructQuote/CheckFunctionTest2.json";
        Json::Value Root;
```

(下页继续)

(续上页)

```
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value _CheckFunctionTest2_Root = Root["CheckFunctionTest2" + std::to_
↪string(times)];
        /*It is the 1 parameter: CheckFunc    CheckFunctionTest2
         *
         * Parameters of the prototype:StructQuote::QuoteTestCase &CheckFunc
         */
        /* CheckFunc */
        Json::Value _CheckFunc_Root = _CheckFunctionTest2_Root["CheckFunc"];
        struct StructQuote::QuoteTestCase _CheckFunc = DriverstructQuoteTestCase(_
↪CheckFunc_Root);
        //The Function of Class    Call
        _StructQuote->CheckFunctionTest(_CheckFunc);
        return 0;
}
```

12.4 结构体引用作为函数返回值

结构体引用作为函数返回值同参数赋值一样，可以张家界调用普通结构体类型的 Return 函数进行赋值。

```
void DriverStructQuote::ReturnDriver_CheckNameXX3(StructQuote::QuoteTestCase& returnType)
{
        const char* JsonFilePath = "drivervalue/StructQuote/CheckNameXX3.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(JsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value CheckNameXX3_Root = Root["CheckNameXX3" + std::to_
↪string(CheckNameXX3Times)];
        /* returnType */
        Json::Value returnType_Root;
        returnType_Root = Returnstruct_QuoteTestCase(returnType_Root, returnType);
        CheckNameXX3_Root["returnType"] = returnType_Root;
        Root["CheckNameXX3" + std::to_string(CheckNameXX3Times)] = CheckNameXX3_Root;
        std::ofstream JsonFile;
        Json::StyledWriter sw;
        JsonFile.open(JsonFilePath);
```

(下页继续)

(续上页)

```
        JsonFile << sw.write(Root);
        JsonFile.close();
}
```

12.5 构造函数中对结构体引用类型的成员变量的赋值

构造函数中赋值部分调用的也是参数赋值部分的内容，所以在结构体引用部分调用的还是普通结构体的父子函数。

```
DriverStructQuote::DriverStructQuote(Json::Value Root, int times)
{
        Json::Value _StructQuote_Root = Root["StructQuote" + std::to_string(times)];
        /* CheckFunc */
        Json::Value _CheckFunc_Root = _StructQuote_Root["CheckFunc"];
        int _CheckFunc_len = _CheckFunc_Root.size();
        StructQuote::QuoteTestCase* _CheckFunc = DriverstructQuoteTestCasePoint(_
↪CheckFunc_Root, _CheckFunc_len);
        /* CheckFuncR */
        Json::Value _CheckFuncR_Root = _StructQuote_Root["CheckFuncR"];
        struct StructQuote::QuoteTestCase _CheckFuncR = DriverstructQuoteTestCase(_
↪CheckFuncR_Root);
        /* CheckFuncRRR */
        Json::Value _CheckFuncRRR_Root = _StructQuote_Root["CheckFuncRRR"];
        struct StructQuote::QuoteTestCase _CheckFuncRRR = DriverstructQuoteTestCase(_
↪CheckFuncRRR_Root);
        _StructQuote = new StructQuote(_CheckFunc, _CheckFuncR, _CheckFuncRRR, false);
}
```

CHAPTER 13

类的二级指针

13.1 测试的源码

测试的源码中考虑到了类的普通类型、类的一级指针、类的二级指针的参数赋值以及作为返回值，构造函数中类中类的二级指针类型的成员变量的赋值。

```
#include"json/json.h"
#include <iostream>
using namespace std;
#pragma once
namespace PointerCase {
        class PointerPair
        {
        public:
                PointerPair();
                ~PointerPair();

        private:
                int Test;
                char pair;

public:
PointerPair(int Test, char pair, bool Wings):Test(Test), pair(pair)
```

(下页继续)

(续上页)

```
{}
};
        class tooltipPoints
        {
        public:
                tooltipPoints();
                ~tooltipPoints();
        private:
                int a;
                char b;

public:
tooltipPoints(int a, char b, bool Wings):a(a), b(b)
{}
};
        class value_TestClass
        {
        public:
                value_TestClass();
                ~value_TestClass();
        private:
                PointerPair buitinClass;
                PointerPair *PointerClass;
                tooltipPoints **TwoPointerClass;
        public:
                void gawp_pointer_options(PointerPair buitinClass);
                void testing_prng_seed(PointerPair *PointerClass);
                void TestSuite_AddLive(tooltipPoints **TwoPointerClass);
                PointerCase::tooltipPoints TestClassReturn(PointerPair buitinClass);
                PointerCase::tooltipPoints *TestClassReturnPointer(PointerPair␣
↪buitinClass);
                PointerCase::tooltipPoints **TestClassReturnTwoPointer(PointerPair␣
↪buitinClass);
public:
value_TestClass(PointerCase::PointerPair buitinClass, PointerCase::PointerPair␣
↪*PointerClass, PointerCase::tooltipPoints **TwoPointerClass, bool␣
↪Wings):buitinClass(buitinClass), PointerClass(PointerClass),␣
↪TwoPointerClass(TwoPointerClass)
{}
};
```

(下页继续)

(续上页)

```
}
```

13.2 类的二级指针的值生成

类的二级指针直接作为普通类类型进行值生成。

问题：为什么类的二级指针直接作为普通类类型进行值生成?

答：因为在类的二级指针进行赋值时，传递的时引用，所以只需要实例化一次就够了，即 new 一个该类的指针。

```
{
   "PointerPair0" : {
          "Test" : 7443,
          "pair" : "C"
   },
   "tooltipPoints0" : {
          "a" : 3804,
          "b" : "V"
   },
   "value_TestClass0" : {
          "PointerClass" : [
                 {
                        "Test" : 2292,
                        "pair" : "j"
                 }
          ],
          "TwoPointerClass" : {
                 "a" : 5836,
                 "b" : "x"
          },
          "buitinClass" : {
                 "Test" : 2673,
                 "pair" : "k"
          }
   }
}
```

13.3 类的二级指针驱动生成

通过从 json 中取值，然后进行对类的成员变量利用插装的构造函数进行赋值，然后实例化该类，将该类的引用赋给定义的一个类的二级指针，进行传参实现对原函数的调用。

问题：为什么使用引用?

答：因为要调用原函数，我们只需要传一个类的二级指针即可，并且对应值生成部分的内容，只需要 new 一个指针，然后传引用就可以实现目的。

```
int Divervalue_TestClass::Drivervalue_TestClassTestSuite_AddLive2(int times)
{
        TestSuite_AddLive2Times = times;
        /* Root is the json object of the value file.TestSuite_AddLive2_Root is function.
↪TestSuite_AddLive2 is json object.  */
        const char* jsonFilePath = "drivervalue/value_TestClass/TestSuite_AddLive2.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value _TestSuite_AddLive2_Root = Root["TestSuite_AddLive2" + std::to_
↪string(times)];
        /*It is the 1 parameter: TwoPointerClass    TestSuite_AddLive2
         *
         * Parameters of the prototype:PointerCase::tooltipPoints **TwoPointerClass
         */
        Json::Value _TwoPointerClassTwoPointerClass_Root = _TestSuite_AddLive2_Root[
↪"TwoPointerClass"];
        /* a */
        int _TwoPointerClassa = _TwoPointerClassTwoPointerClass_Root["a"].asInt();
        /* b */
        string _TwoPointerClassbStr = _TwoPointerClassTwoPointerClass_Root["b"].
↪asString();
        char _TwoPointerClassb = _TwoPointerClassbStr[0];
        PointerCase::tooltipPoints* TwoPointerClass_value = new␣
↪PointerCase::tooltipPoints(_TwoPointerClassa, _TwoPointerClassb, false);
        PointerCase::tooltipPoints** _TwoPointerClass = &TwoPointerClass_value;
        //The Function of Class    Call
        _value_TestClass->TestSuite_AddLive(_TwoPointerClass);
        return 0;
}
```

13.4 类的二级指针作为返回值

类的二级指针作为返回值调用的是参数捕获中插装的函数（W_MemberVarCaputer()），需要注意的是当没有进行参数捕获代码生成时，是没有该函数的。

```
void Drivervalue_TestClass::ReturnDriver_
↪TestClassReturnTwoPointer5(PointerCase::tooltipPoints** returnType)
{
        const char* JsonFilePath = "drivervalue/value_TestClass/
↪TestClassReturnTwoPointer5.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(JsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value TestClassReturnTwoPointer5_Root = Root["TestClassReturnTwoPointer5"␣
↪+ std::to_string(TestClassReturnTwoPointer5Times)];
        /* returnType */
        TestClassReturnTwoPointer5_Root["returnType"] = returnType[0]->W_
↪MemberVarCaputer();
        Root["TestClassReturnTwoPointer5" + std::to_
↪string(TestClassReturnTwoPointer5Times)] = TestClassReturnTwoPointer5_Root;
        std::ofstream JsonFile;
        Json::StyledWriter sw;
        JsonFile.open(JsonFilePath);
        JsonFile << sw.write(Root);
        JsonFile.close();

}
```

13.5 构造函数中对类的二级指针的成员变量赋值

构造函数中对类的二级指针的成员变量赋值部分调用的是参数赋值的逻辑，所以与之相比并没有什么改变。

```
Drivervalue_TestClass::Drivervalue_TestClass(Json::Value Root, int times)
{
        Json::Value _value_TestClass_Root = Root["value_TestClass" + std::to_
↪string(times)];
        Json::Value _buitinClassbuitinClass_Root = _value_TestClass_Root["buitinClass"];
        /* Test */
        int _buitinClassTest = _buitinClassbuitinClass_Root["Test"].asInt();
```

(下页继续)

(续上页)

```
        /* pair */
        string _buitinClasspairStr = _buitinClassbuitinClass_Root["pair"].asString();
        char _buitinClasspair = _buitinClasspairStr[0];
        PointerCase::PointerPair _buitinClass(_buitinClassTest, _buitinClasspair, false);
        int _PointerClasspointSize = 0;
        Json::Value _PointerClassPointerClass_Root = _value_TestClass_Root["PointerClass
↪"][_PointerClasspointSize];
        /* Test */
        int _PointerClassTest = _PointerClassPointerClass_Root["Test"].asInt();
        /* pair */
        string _PointerClasspairStr = _PointerClassPointerClass_Root["pair"].asString();
        char _PointerClasspair = _PointerClasspairStr[0];
        PointerCase::PointerPair* _PointerClass = new PointerCase::PointerPair(_
↪PointerClassTest, _PointerClasspair, false);
        Json::Value _TwoPointerClassTwoPointerClass_Root = _value_TestClass_Root[
↪"TwoPointerClass"];
        /* a */
        int _TwoPointerClassa = _TwoPointerClassTwoPointerClass_Root["a"].asInt();
        /* b */
        string _TwoPointerClassbStr = _TwoPointerClassTwoPointerClass_Root["b"].
↪asString();
        char _TwoPointerClassb = _TwoPointerClassbStr[0];
        PointerCase::tooltipPoints* TwoPointerClass_value = new␣
↪PointerCase::tooltipPoints(_TwoPointerClassa, _TwoPointerClassb, false);
        PointerCase::tooltipPoints** _TwoPointerClass = &TwoPointerClass_value;
        _value_TestClass = new PointerCase::value_TestClass(_buitinClass, _PointerClass,␣
↪_TwoPointerClass, false);
}
```

CHAPTER 14

类的运算符重载

逻辑思想：

运算符重载函数分为成员函数重载和非成员函数重载（即 friend），两者的驱动生成会有所不同，主要表现在调用原函数上；首先在函数中将运算符重载函数分离出来；再将运算符重载函数区分为非成员函数（friend）和成员函数；然后分别进行具体的运算符进行驱动生成，不同的运算符会有不同处理。

问题：为什么要将运算符重载函数与普通函数分离?

答：在生成驱动时，运算符重载成员函数与类的普通成员函数区别在于类名以及参数，前者会有一个运算符的后缀，以及在成员函数的运算符重载中，有一个 this 指针隐含的传过去。

问题：为什么要将运算符重载函数区分为非成员函数（friend）和成员函数?

答：因为 friend 和普通函数的处理会有不同，主要表现在非成员重载函数正常会比成员重载函数多一个参数，成员函数的运算符重载中，有一个 this 指针隐含的传过去，以及在调用部分也会不同，成员重载函数可以直接使用类指针进行调用，而 friend 不一样。

问题：为什么要对不同的操作符进行不同的处理?

答：因为操作符直接的差异很大，有的并不需要参数，而需要参数的一部分是人为赋值的参数，一部分是系统参数，需要分开处理。

14.1 测试的源码

```
#pragma warning(disable:4996)
#include"json/json.h"
#include <iostream>
#include <fstream>
using namespace std;
#pragma once
class ClassOpertor
{
public:
        ClassOpertor();
        ~ClassOpertor();
        ClassOpertor operator- ();
        bool operator<(const ClassOpertor& d);
        ClassOpertor operator+(ClassOpertor&add);
        ClassOpertor operator+(int b);
        void displayDistance();
        friend ClassOpertor operator+(const int b, ClassOpertor obj);
        friend ClassOpertor operator+(const ClassOpertor& a, const ClassOpertor& b);
        ostream &operator<<(ostream &output);
        friend ostream &operator<<(ostream &output, const ClassOpertor &Dis);
        ClassOpertor operator++();//前缀形式
        ClassOpertor operator++(int);//后缀形式
        int fun();
        bool operator!();
        void * operator new(size_t size);//一个参数，为 size_t 类型
        void operator delete(void *ptr);// void 类型的指针作为参数
        void operator=(const ClassOpertor &D);
        void operator+=(const ClassOpertor &D);
        int& operator[](int i);//下标
        ClassOpertor& operator*();//指针
        ClassOpertor* operator&();//取地址符
        int operator()(int val);
        ClassOpertor* operator->();
        void operator delete[](void*, size_t size);
private:
        int length;
        int weight;
        int arr[10];
        ClassOpertor *persion;
public:
```

(下页继续)

(续上页)

```
ClassOpertor(int length, int weight, int arr[10], bool Wings):length(length),␣
↪weight(weight)
{
        /* arr */
   for(unsigned int size = 0; size < 10; size++)
   {
           this->arr[size] = arr[size];
   }
}
};
```

14.2 成员函数的处理（篇幅原因，每种只列标志性的运算符重载的驱动代码）

1. 算数运算符

+、-、*、/、%、++、–

加法（+）测试源码：

```
Operatortor Operatortor::operator+=(int len)
{
        length += len;
        return Operatortor();
}
```

加法（+）驱动：

```
int DriverClassOpertor::DriverClassOpertoroperator2(int times)
{
        operator2Times = times;
        const char* jsonFilePath = "drivervalue/ClassOpertor/operator2.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value _operator2_Root = Root["operator2" + std::to_string(times)];
        Json::Value _addadd_Root = _operator2_Root["add"];
        /* length */
        int _addlength = _addadd_Root["length"].asInt();
```

(下页继续)

(续上页)

```
        /* weight */
        int _addweight = _addadd_Root["weight"].asInt();
        /* arr */
        int _addarr[10];
        for (int len = 0; len < 10; len++) {
                _addarr[len] = _addadd_Root["arr"][len].asInt();
        }
        ClassOpertor _addadd(_addlength, _addweight, _addarr, false);
        _ClassOpertor->operator+(_addadd);
        return 0;
}
```

前置自增（++length）测试源码：

```
Operatortor Operatortor::operator++()
{
        ++length;
        if (length >= 60)
        {
                ++weight;
                length -= 60;
        }
        return ClassOpertor();
}
```

前置自增（++length）驱动：

```
int DriverOperatortor::DriverOperatortoroperator8(int times)
{
        _ClassOpertor->operator++();
        return 0;
}
```

2. 关系运算符

<、>、==、>=、<=、!=

小于号（<）源码：

```
bool Operatortor::operator<(const Operatortor & d)
{
        if (length < d.length)
```

(下页继续)

(续上页)

```
        {
                return true;
        }
        if (length == d.length && weight < d.weight)
        {
                return true;
        }
        return false;
}
```

小于号（<）驱动：

```
int DriverOperatortor::DriverOperatortoroperator1(int times)
{
        operator1Times = times;
        const char* jsonFilePath = "drivervalue/Operatortor/operator1.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value _operator1_Root = Root["operator1" + std::to_string(times)];
        Json::Value _d_Root = _operator1_Root["d"];
        /* length */
        int _dlength = _d_Root["length"].asInt();
        /* weight */
        int _dweight = _d_Root["weight"].asInt();
        Operatortor _dd(_dlength, _dweight, false);
   // The Function of Class    Call
        _ClassOpertor->operator<(_dd);
        return 0;
}
```

小于等于号（<=）驱动：

```
int DriverOperatortor::DriverOperatortoroperator1(int times)
{
        operator1Times = times;
        const char* jsonFilePath = "drivervalue/Operatortor/operator2.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
```

(下页继续)

(续上页)

```
        _reader.parse(_ifs, Root);
        Json::Value _operator2_Root = Root["operator2" + std::to_string(times)];
        Json::Value _d_Root = _operator2_Root["d"];
        /* length */
        int _dlength = _d_Root["length"].asInt();
        /* weight */
        int _dweight = _d_Root["weight"].asInt();
        Operatortor _dd(_dlength, _dweight, false);
   // The Function of Class    Call
        _ClassOpertor->operator<=(_dd);
        return 0;
}
```

3. 赋值运算符

-=、+=、/=、*=、%=、&=、^=、|=、<<=、>>=、=

= 赋值运算符

```
int DriverClassOpertor::DriverClassOpertoroperator12(int times)
{
        operator12Times = times;
        const char* jsonFilePath = "drivervalue/ClassOpertor/operator12.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value _operator12_Root = Root["operator12" + std::to_string(times)];
        Json::Value _DD_Root = _operator12_Root["D"];
        /* length */
        int _Dlength = _DD_Root["length"].asInt();
        /* weight */
        int _Dweight = _DD_Root["weight"].asInt();
        /* arr */
        int _Darr[10];
        for (int len = 0; len < 10; len++) {
                _Darr[len] = _DD_Root["arr"][len].asInt();
        }
        ClassOpertor _DD(_Dlength, _Dweight, _Darr, false);
        //The Function of Class    Call
        _ClassOpertor->operator=(_DD);
```

(下页继续)

(续上页)

```
        return 0;
}
```

+= 赋值运算符

```
int DriverClassOpertor::DriverClassOpertoroperator13(int times)
{
        operator13Times = times;
        const char* jsonFilePath = "drivervalue/ClassOpertor/operator13.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value _operator13_Root = Root["operator13" + std::to_string(times)];
        Json::Value _DD_Root = _operator13_Root["D"];
        /* length */
        int _Dlength = _DD_Root["length"].asInt();
        /* weight */
        int _Dweight = _DD_Root["weight"].asInt();
        /* arr */
        int _Darr[10];
        for (int len = 0; len < 10; len++) {
                _Darr[len] = _DD_Root["arr"][len].asInt();
        }
        ClassOpertor _DD(_Dlength, _Dweight, _Darr, false);
        //The Function of Class    Call
        _ClassOpertor->operator+=(_DD);
        return 0;
}
```

4. 单目运算符

+（正）、-（负）、*（指针）、&（取地址）

&(取地址)

```
int DriverClassOpertor::DriverClassOpertoroperator16(int times)
{
        operator16Times = times;
        const char* jsonFilePath = "drivervalue/ClassOpertor/operator16.json";
        Json::Value Root;
        Json::Reader _reader;
```

(下页继续)

(续上页)

```
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value _operator16_Root = Root["operator16" + std::to_string(times)];
        _ClassOpertor->operator&();
        return 0;
}
```

-（负）运算符

```
int DriverClassOpertor::DriverClassOpertoroperator0(int times)
{
        _ClassOpertor->operator-();
        return 0;
}
```

5. 逻辑运算符和位运算符

&、|、^、&&、||、!、<<(左移)、>>(右移)、~（按位取反）

&& 和 || 不建议重载运算符。

问题： 为什么不建议重载 && 和 ||?

答： 逻辑 && 和逻辑 || 运算符是可以重载的，但是重载不能实现逻辑 && 和逻辑 || 运算符的短路功能。

```
!=（逻辑非运算符）
int DriverClassOpertor::DriverClassOpertoroperator9(int times)
{
        operator9Times = times;
        const char* jsonFilePath = "drivervalue/ClassOpertor/operator9.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value _operator9_Root = Root["operator9" + std::to_string(times)];
        _ClassOpertor->operator!();
        return 0;
}
```

6. 流运算符

<<、>>

对于 C++ 的输入输出流的重载函数来说，参数 output 我们在驱动赋值时并不需要去定义和赋值，可以直接使用 cout，即标准 c++ 的输出函数。

<< 输出流的源码：

```
ostream & ClassOpertor::operator<<(ostream & output)
{
        return output;
}
<<输出流运算符
int DriverClassOpertor::DriverClassOpertoroperator5(int times)
{
        _ClassOpertor->operator<<(cout);
        return 0;
}
```

7. 空间申请与释放

new、delete、new[]、delete[]

new 运算符

```
int DriverClassOpertor::DriverClassOpertoroperator10(int times)
{
        operator10Times = times;
        const char* jsonFilePath = "drivervalue/ClassOpertor/operator10.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value _operator10_Root = Root["operator10" + std::to_string(times)];
        /* size */
        unsigned int _size = _operator10_Root["size"].asUInt();
        _ClassOpertor->operator new(_size);
        return 0;
}
```

delete 运算符

```
int DriverClassOpertor::DriverClassOpertoroperator11(int times)
{
        operator11Times = times;
        const char* jsonFilePath = "drivervalue/ClassOpertor/operator11.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
```

(下页继续)

(续上页)

```
        _reader.parse(_ifs, Root);
        Json::Value _operator11_Root = Root["operator11" + std::to_string(times)];
        /* ptr */
        void* _ptr = nullptr;
        //The Function of Class    Call
        _ClassOpertor->operator delete(_ptr);
        return 0;
}
```

8. 特殊运算符

->（类成员访问运算符）、()（函数运算符）、[]（下标运算符）、co_await、,(逗号))

[] 下标运算符源码

```
int & ClassOpertor::operator[](int i)
{
        int SIZE = 10;
        if (i > SIZE)
        {
                cout << "索引超过最大值" << endl;
                return arr[0];
        }
        return arr[i];
}
```

[] 下标运算符驱动

```
int DriverClassOpertor::DriverClassOpertoroperator14(int times)
{
        operator14Times = times;
        const char* jsonFilePath = "drivervalue/ClassOpertor/operator14.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value _operator14_Root = Root["operator14" + std::to_string(times)];
        /* i */
        int _i = _operator14_Root["i"].asInt();
        _ClassOpertor->operator[](_i);
        return 0;
}
```

() 函数运算符

```
int DriverClassOpertor::DriverClassOpertoroperator17(int times)
{
        operator17Times = times;
        const char* jsonFilePath = "drivervalue/ClassOpertor/operator17.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value _operator17_Root = Root["operator17" + std::to_string(times)];
        /* val */
        int _val = _operator17_Root["val"].asInt();
        _ClassOpertor->operator()(_val);
        return 0;
}
```

->(类成员访问符)

```
int DriverClassOpertor::DriverClassOpertoroperator9(int times)
{
        _ClassOpertor->operator->();
        return 0;
}
```

,（逗号运算符）

```
int DriverClassOpertor::DriverClassOpertoroperator2(int times)
{
        operator2Times = times;
        const char* jsonFilePath = "drivervalue/ClassOpertor/operator18.json";
        Json::Value Root;
        Json::Reader _reader;
        std::ifstream _ifs(jsonFilePath);
        _reader.parse(_ifs, Root);
        Json::Value _operator2_Root = Root["operator2" + std::to_string(times)];
        Json::Value _rdd_Root = _operator2_Root["add"];
        /* length */
        int _rddlength = _rdd_Root["length"].asInt();
        /* weight */
        int _rddweight = _rdd_Root["weight"].asInt();
        /* arr */
```

(下页继续)

(续上页)

```
        int _addarr[10];
        for (int len = 0; len < 10; len++) {
                _rddarr[len] = _rdd_Root["arr"][len].asInt();
        }
        ClassOpertor _rdd(_rddlength, _rddweight, _rddarr, false);
        _ClassOpertor->operator->(_rdd);
        return 0;
}
```

14.3 非成员函数（friend）的处理

非成员的重载函数在驱动生成上和成员函数区别并不是很大，主要表现在以下两点：

1）函数参数上，作为成员函数重载时，，有一个 this 指针隐含的传过去，不需要驱动中给它额外赋值；非成员的重载函数则没有，相同符号的重载，后者会比前者多一个参数，这里需要驱动给它赋值。

2）调用原函数上，作为成员函数重载时，可以使用类指针进行调用重载函数，非成员的重载函数则不需要，可以直接使用。

CHAPTER 15

void 和 void 功能函数指针使用

15.1 void

1. 打开工程会进行 xml 读取

```
WingsXmlInfoStorage::OpenSpecialFile()
```

函数会对 void.xml 和 functionPointer.xml 进行读取，

将读取的值存放在 vector 容器中。

2. 点击特殊赋值中的 void* 设置

如果文件夹中不存在 void.xml, 则在此处生成 void.xml

3. 根据 1 步骤中存放含有 void.xml 的容器

```
void WingsShowVoidInfoWidget::StartCreateVoidInfoTable(QStandardItemModel *model_value,␣
↪QString Path)
```

在此函数执行后生成，生成树状表格

4. 在 void* 界面中会有改工程中的所需要填写的 void* 的真正类型

5. 在 reference 栏中填写正在的类型，点击保存，即可生成 void.xml 文件。在文件中可以看到保存的真实类型。

```
void on_pushButton_save_clicked();
```

保存完后，会重新执行 xmltostorage，读取 xml，更新保存 void 的容器中的信息

6. 执行 VoidPointerReplace() 函数。

遍历保存 void 的容器，和 recordxml 容器里面的信息，对比它们属于的的类名或者函数名

如果参数不是结构体真实值保存在 voidPointerVar 中，

如果参数是结构体或者类，存放在 paramChildInfo 中的是整个结构体或者类结构上所有的 void* 的信息

7. 在进行驱动生成等操作时候，检测参数类型是否为 void*。

如果参数为 void*，将参数上的 voidPointerVar 替换原本的 typeinfo

如果该参数为结构体其中含有 void*，其中 paramChildInfo 中保存的是这个结构体或者类的全部 void* 的真实信息，需要进行处理，将结构体中的每个参数的 paramChildInfo 都存放上属于自己的真实类型

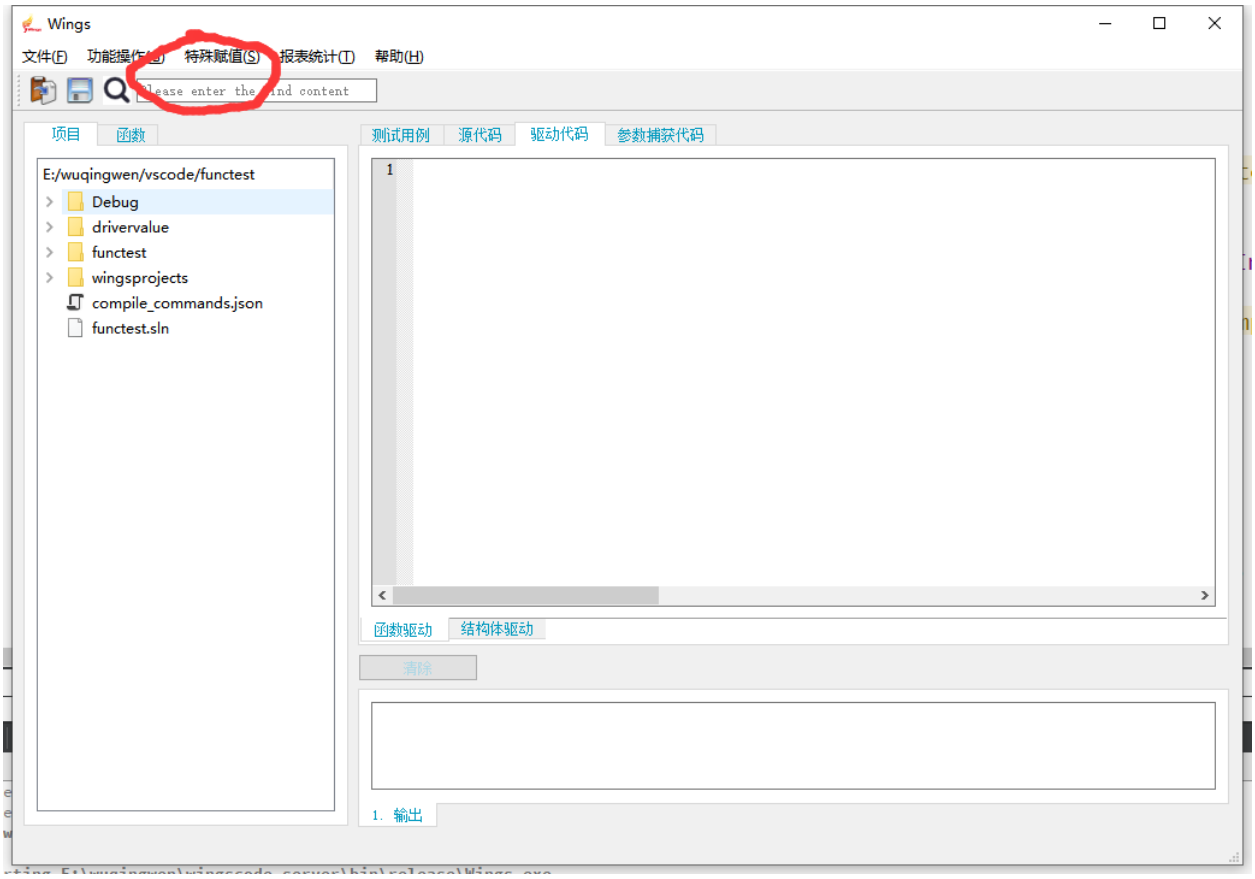
FuncChilderVecReplace

该函数用来处理这种情况

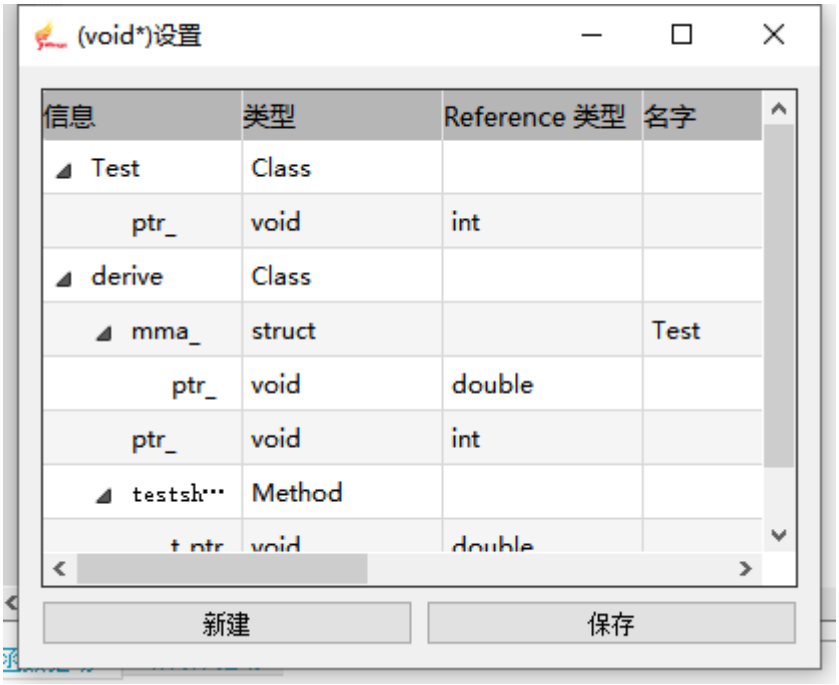
注: 函数指针处理方式和 void* 相同

15.2 void* 功能使用

1. 打开工程后点击特殊赋值，选中特殊赋值栏中的（void*）设置

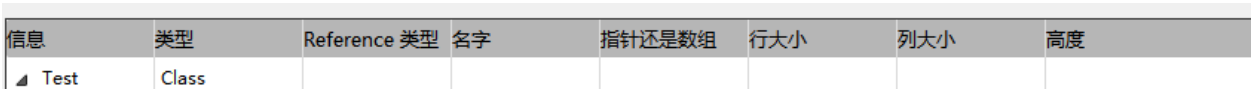


2. 在界面中配置 void* 的类型



在途中 Reference 类型栏中填写 void* 的真正类型

在这里可以配置数组的维数长度，最多可使用三维数组



3. 配置完成后点击保存按键，此时 void.xml 中以及保存了之前配置的基本信息

```
<Test ClassRecord="Class">
    <ptr_ type="ZOA_VOID" baseType1="PointerType">
        <ReferenceType baseType1="PointerType" type="ZOA_INT" />
    </ptr_>
</Test>
<derive ClassRecord="Class">
    <mma_ type="StructureOrClassType" name="Test" baseType1="BuiltinType">
        <ptr_ type="ZOA_VOID" baseType1="PointerType">
            <ReferenceType baseType1="PointerType" type="ZOA_DOUBLE" />
        </ptr_>
    </mma_>
    <ptr_ type="ZOA_VOID" baseType1="PointerType">
        <ReferenceType baseType1="PointerType" type="ZOA_INT" />
    </ptr_>
    <testshow0 Method="Method">
        <t_ptr type="ZOA_VOID" baseType1="PointerType">
            <ReferenceType baseType1="PointerType" type="ZOA_DOUBLE" />
        </t_ptr>
    </testshow0>
</derive>
```

4. 配置完成后，点击驱动生成。在以及配置过的类或者函数中，void* 将被替换为配置好的值进行驱动生成

或者值生成

例如：此处的 Test 中的 ptr_ 成员变量就会按照配置好的类型 int 进行值生成

```
"Test0" : {
    "ptr_" : [ 2428, 3456, 6110 ]
},
```

注释：函数指针配置方式相同

CHAPTER 16

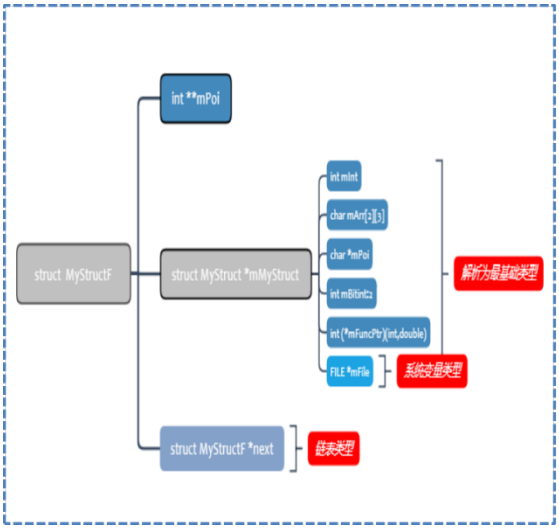
数据表格

Wings 目前测试用例数据采用随机生成的方式，支持 int、char、double、float、bool、char* 类型。数据表格可以任意编辑以上类型的数值。

(1) wings 数据表格将会针对参数进行展开，假如参数类型为结构类型，数据表格将分层展开结构的类型，到基本类型。

(2) 针对基本类型的指针类型，例如 int *p;wings 处理为不定长度的一维数组类型，int **p; 处理为不定长度的二维数组类型，默认长度为 3，数据表格界面可以点击进行添加和删除数据。

(3) 针对不定长度的数组作为函数参数，例如 int p[];wings 默认长度为 1，用户依据需求，在数据表格界面进行任意添加和修改即可。



func1		
myf	struct MyStructF *	
[0]		
mPoi	int **	
mMyStruct	struct MyStruct *	
[0]		
mInt	int	7385
mArr	char [][]	
[0]		TUI
[1]		YNM
mPoi	char *	H97
mBitInt	int	31690
mFuncPtr	int(*)(int,double)	NULL
mFile	struct _iobuf *	{DblPv3,rb+}
[1]		
[2]		
next	struct MyStructF * (->)	
mPoi	int **	
mMyStruct	struct MyStruct *	
next	struct MyStructF * (->)	NULL
[1]		
[2]		
b	int	10624
ggloMys	struct GlobeMyStruct	
gInt	int	14740
return	int	0
expectreturn	int	0

CHAPTER 17

wings 部署操作说明

wings 目前支持 windows 与 linux 平台。

17.1 系统配置要求

wings 客户端目前支持 windows、linux 操作系统平台。机器的最低配置如下：

CPU	内存	硬盘
4 核	8G	100G

17.2 编译数据库

17.2.1 linux 下编译数据库的生成

（1）针对 makefile 的工程，星云测试提供安装的 bear 的安装包，按照步骤操作即可。安装 bear https://github.com/rizsotto/Bear（bear 所需 python>2.7）cmake

（3）类似的其他生成编译数据库的方式，可以参考 https://sarcasm.github.io/notes/dev/compilation-database.html。

（2）针对 cmake 工程，设置-DCMAKE_EXPORT_COMMANDS=ON 即可，会在对应的 build 目录下生成 compile_commands.json 文件。

17.2.2 windows 下编译数据库的生成

（1）windows 下的目前支持 vs2015 以上版本与 Msys2+mingw 的编译环境。

（2）安装星云测试提供的 vs 插件，Sourcetrail.Extension.v2.0.3.130.vsix 双击安装之后，会在 vs 的菜单栏出现 Sourcetrail，选择 Create Compilation Database，生成对应源程序的 complie_command.json 文件。

（3）qt 中包含有生成自带的编译数据库的文件，在编辑->Generate Copmliation Database for “工程”