Wings Documentation

Release 1.0.0

TeststarsTeam

Jul 20, 2020

Contents

CHAPTER 1

Introduction to Wings

After the system integration is completed, wings can be used to generate driver code, parameter capture, and playback for any level of functions. Low-level functions do unit testing, and the high-level functions do interface and system testing.It can use the system test data as input for verification.The application of this technology will enable the recording and verification of automated tests to be carried out almost without human involvement.

1.1 Case

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

1.2 Download

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

1.3 Support

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

CHAPTER 2

Unit testing automatic generation technology

2.1 Features of wings automatic generation technology

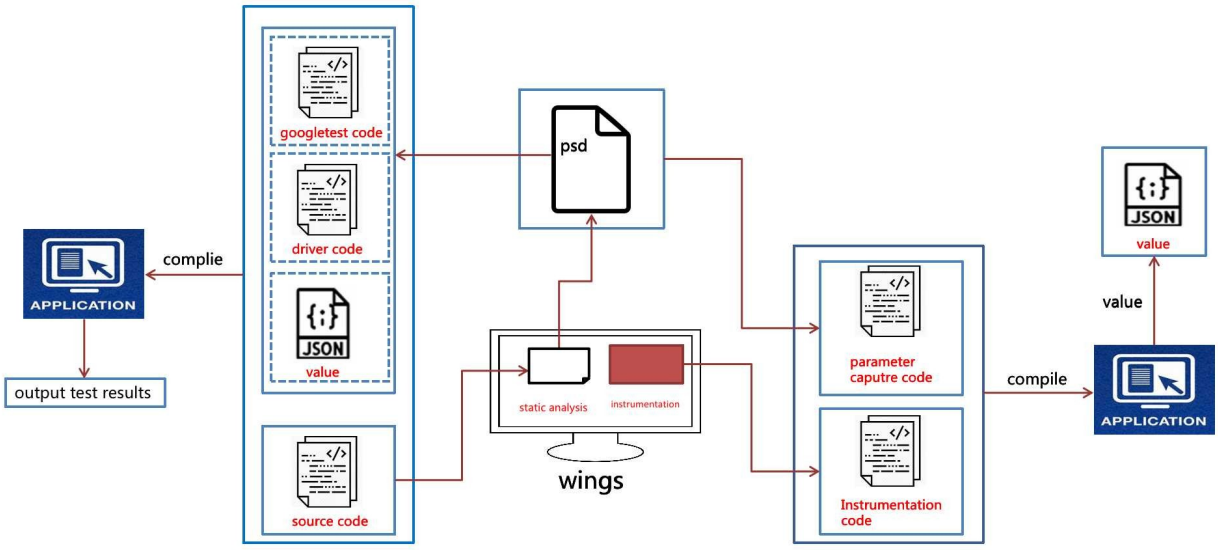
- Wings is an intelligent fully automatic test case driven generation system, and can automatically generate test drivers and parameter capture programs.
- Wings supports the C/C ++ language and can handle any type (built-in type and complex types) of parameters, global variables, and return values.
- Supports visual data tables for assignment, regardless of the driver itself. Data tables can express data relationships at any depth and level, and users only need to edit the table data.Ability to distinguish between system types and custom types.
- Support special template assignment for complex system types, such as std::vector, rather than expanding all complex system variables.

2.2 The overall framework of wings

First of all, wings uses static analysis technology to extract the backbone information of the program under test. The main steps are as follows:

- Get compilation database file for different platforms.
- According to the compiled database file, the static analysis of the source code, the extracted information is stored in the PSD structure.

- Read the PSD structure to generate the driver code, GoogleTest code, and the value file. The generated code compiles with the source code. Finally, the test results are output.
- Read the PSD structure to generate the parameter capture code . The generated code is compiled with the source code to get the parameter values of the program at run time.



CHAPTER 3

Parameter Structure Description(PSD)

PSD refers to the description of the source code, including class name, class member variable information, class function information, and various types of information.Type information refers to the data types of function parameters, global variables, and return values. Complex data types are expanded layer by layer to built-in types (char, int, string, and so on).The PSD structure is stored in an XML file, and different XML files store different description information.

RecordDecl.xml: store the analysis and development results of structures ,unions and classes in the entire project.

EnumDecl:store the enumerate information in the entire project.

filename.xml : store function information for each file.

A brief description of some attributes is as follows:

Type attribute

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	built-in type
StructureOrClassType	Structures type
ZOA_FUNC	function pointer type
ZOA_UNION	union type
ZOA_ENUM	enumeration type
ClassType	class type

Basetype attribute

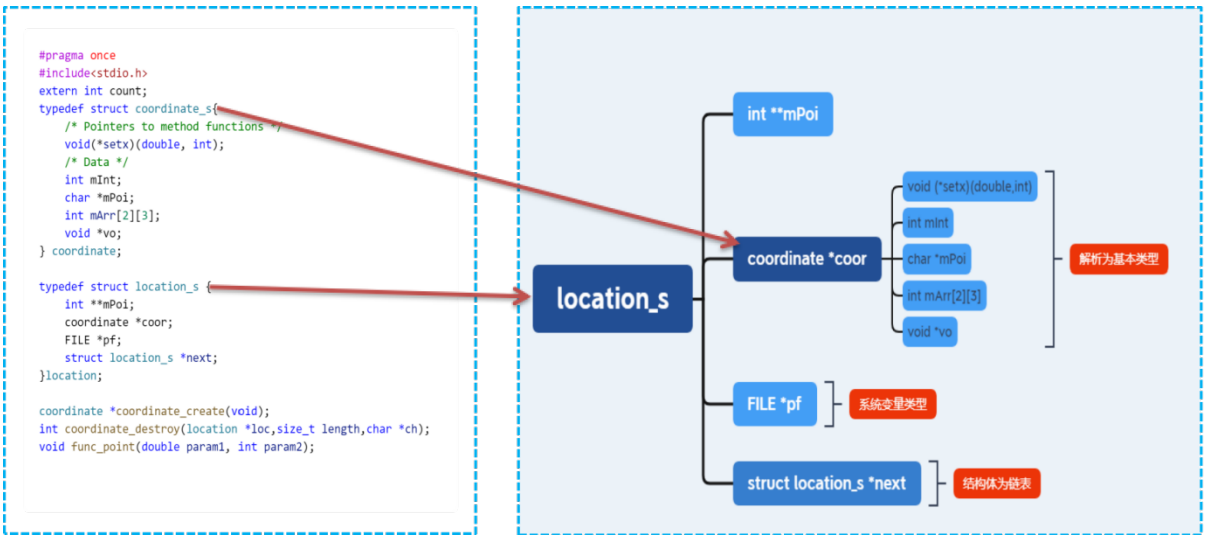
BuiltinType	built-in type
ArrayType	array type
PointerType	pointer type
StructureOrClassType	Structures type
UnionType	union type
EnumType	enumeration type
FunctionPointType	function pointer type

Other attributes

Name	Represents the names of structures, classes, and unions
NodeType	linked lists
parmType	function parameter
parNum	The number of function
SystemVar	type in system include
value	The value of the enumeration type
bitfield	Bytes occupied by bitfield
returnType	return type
Field	member of class
Method	class constructor
paramName	class constructor parameter name
paramType	class constructor parameter types
TemplateArgumentType	Template parameter type
TemplateArgumentValue	The parameters in the structure are concrete values
FunctionModifiers	function access
FunctionAttribute	Function is extern or static function
FuncClassName	The class to which the function belongs
OperatorFundecl	Overload operator functions
Operator	Overload operator types

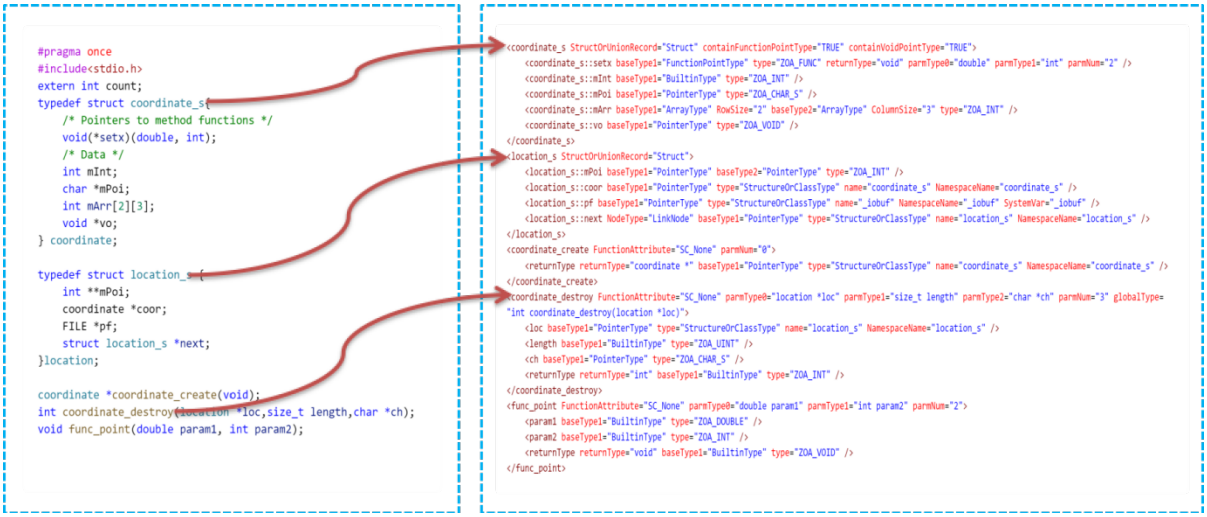
3.1 C language PSD structure description

For complex types, such as the structure type location_s , in addition to the basic data type, the member variable also contains the structure type. In the code shown in the following figure 2, the location_s contains the coordinate_s structure, and FILE and other types of information To distinguish between different types.



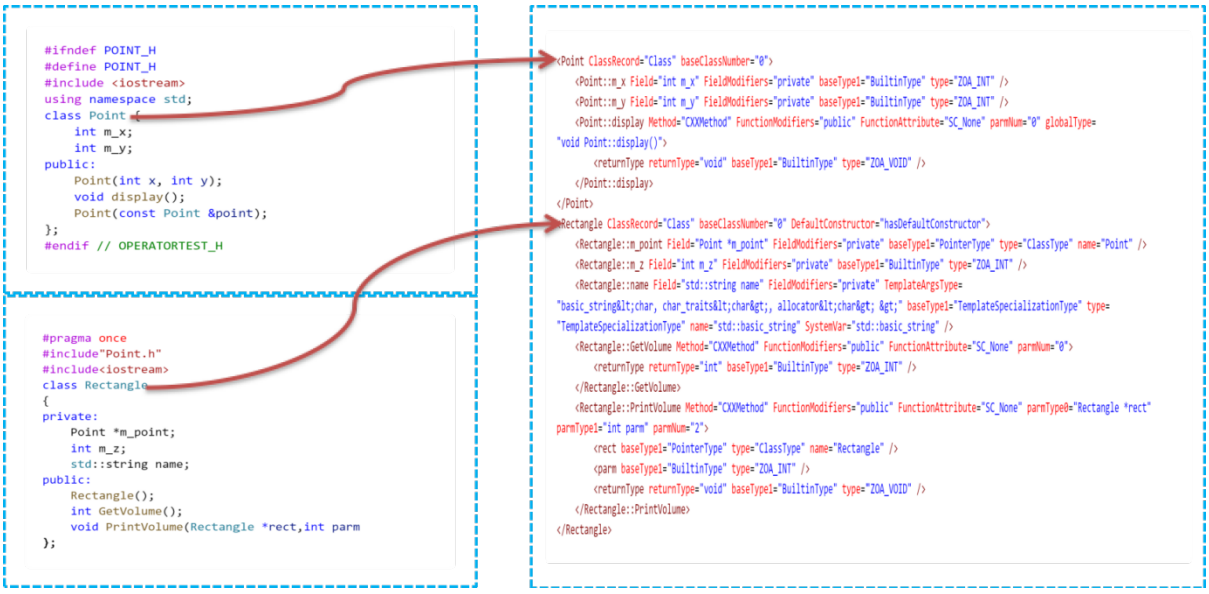
The PSD storage structure of FIG . 2 above is shown in FIG. 3 , where the description information of

the structure mainly includes member variable names, types of member variables, and information such as judging whether the member variable is a system variable or a linked list. For different information, different information processing is done during driver generation or parameter capture.



3.2 C ++ PSD structure description

C ++ mainly indicates that the type is a class, so the test is that c ++ takes a class as a unit for testing. The class mainly includes the member variable name and type information of the class, and the access permission information of the member variable. The member functions of the class are divided into constructors, inline functions, virtual functions, etc., parameter information and type information of member functions.



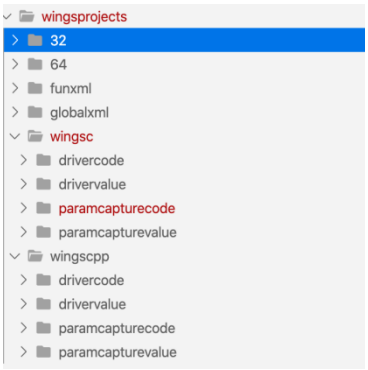
CHAPTER 4

Introduction unit testing code

4.1 C unit test

4.1.1 Naming rules (using Google C++ Style Guide)

1. The generated file directory is shown in Figure 5 below:



The unit test code is stored in the drivercode folder. 2. driver.cc and driver.h mainly store some public functions and header files in the program. 3. The same structure or union may be used as the parameters of multiple functions. In order to avoid code duplication, wings are encapsulated into different drive functions or parameter capture functions for all types of structures and unions.

Examples are as follows:

For the structure information in figure 2, 5 different driver function information will be generated, as shown in below:

```
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);
```

The naming rules for structure implementation functions are: DriverStruct+ struct name + type, where Point represents a primary pointer or a one-dimensional array, and PointPoint represents a secondary pointer or a two-dimensional array.For each function, the corresponding driver function is generated.

The naming rules for source files are: **driver_** + source file name + .cc

For example: driver_nginx.cc

Driver function naming rules: **Driver_** + function name

For example: Driver_ngx_show_version_info (void);

Note: For static test function, it is necessary to remove the static functions Add the declaration of the source function before the driver function.

For example:extern void ngx_show_version_info(void); Driver_ngx_show_version_info(void);

5. output of return value The naming rule for the printout function of the return value: Driver + Return + **Print_** + function name. For example: DriverReturnPrint_ngx_show_version_info ();

6.The main function in the user source code needs to be commented out manually. Wings will regenerate a main function file in the driver code for testing. Wings will generate the main function file that drives main as : gtest_auto_main.cc Note: Users choose to use it according to their needs.

4.1.2 Generate code

Function prototype: int coordinate_destory (location_s * loc, size_t length, char * ch) The type of the return value is int , the type of the first parameter is location * , the type of the second parameter is size_t , and the type of the third parameter is char * Assign values to the above three parameters.

The complete drive code for the coordinate_destory function is described in detail in below .

```
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";
```

(continues on next page)

(continued from previous page)

```
    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;
    /*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);
```

(continues on next page)

(continued from previous page)

```
        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++)
            {
                _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 */
```

(continues on next page)

(continued from previous page)

```
        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);

        _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 ++ unit testing

C ++ is mainly for testing each class, wings will automatically generate the driver code of each class and the corresponding GoogleTest code.

4.2.1 Naming rules

The file name corresponding to the driver generated by each class is: driver + class name.cc and .h , the name of the driver class is: Driver + ClassName (class name) , there are many operator overloaded functions in c ++ , in order to ensure the uniqueness of the function, The functions for the class are numbered from 0 , that is, Driver + function name + number.

4.2.2 Generate code

The corresponding test class code will be generated for each class in C ++ . As shown in below, Rectangle is the class under test and contains 3 member variables of type Point , int , std :: string, assuming Rectangle as

the function parameter, the configuration of such object requires . 3 members variable assignment, WINGS automatically inserts a constructor in the source code for each class, the member variable assignment.

```
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;
        }
};
```

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

For each driver class in below, a corresponding constructor will be generated to initialize the tested class, as shown in below :

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

In the driver class, a driver function is generated for each function. As shown in below :

```
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();
```

(continues on next page)

(continued from previous page)

```
  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;
}
```

CHAPTER 5

Introduction to GoogleTest code

In order to compare whether the return value meets the expected output, wings combines the test framework of GoogleTest to automatically complete the writing of the return value and the expected output code.

5.1 C language GoogleTest code

gtest_auto_main.cc this file is the entry file for calling GoogleTest . The content is shown in below :

```
#define _CRT_SECURE_NO_WARNINGS
#include "gtest/gtest.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
  testing::InitGoogleTest(&argc, argv);
  RUN_ALL_TESTS();
  // Start();
  system("pause");
  return 0;
}
```

For each test .c file, a corresponding **driver_** filename_gtest.cc file for function is generated, and the expected comparison operation is performed on the return value of the function coordinate_destory.

In below , the return value of the acquisition function is return , and the expected return value filled in by the user is expected .below is the value file of the measured function.

```
TEST(wings_c_demo_coordinates, coordinate_destroy)
{
        for (int times = 0; times < COORDINATE_DESTROY_INT_TIMES; times++)
        {
                Drive_coordinate_destroy_int(times);
                int coordinate_destroy_int_len = strlen("coordinate_destroy_int");
                char *coordinate_destroy_int_sp = (char *)malloc(sizeof(char) *␣
↪(coordinate_destroy_int_len + 2));
                sprintf(coordinate_destroy_int_sp, "coordinate_destroy_int%d", times);
                const char *jsonFile = "drivervalue/wings_c_demo_coordinates/coordinate_
↪destroy_int.json";
                char *wings_c_demo_coordinates_coordinate_destroy_int_json_data = get_
↪json_data(jsonFile);
                cJSON *Root = cJSON_Parse(wings_c_demo_coordinates_coordinate_destroy_
↪int_json_data);
                cJSON *coordinate_destroy_int_Root = cJSON_GetObjectItem(Root,␣
↪coordinate_destroy_int_sp);
                free(coordinate_destroy_int_sp);
                int _return_actual = cJSON_GetObjectItem(coordinate_destroy_int_Root,
↪"return")->valueint;
                int _return_expected = cJSON_GetObjectItem(coordinate_destroy_int_Root,
↪"expect")->valueint;
                /* return_expected */
                EXPECT_EQ(_return_expected, _return_actual);
        }
}
```

```
{
   "coordinate_destroy_int0" : {
          "ch" : "JTX",
          "count" : 8201,
          "expected" : 10,
          "length" : 2326,
          "loc" : [
                 {
                        "coor" : [
                           {
                                  "mArr" : [
                                         [ 6773, 2513, 7989 ],
                                         [ 7902, 5447, 4144 ]
```

(continues on next page)

(continued from previous page)

```
                                ],
                                "mInt" : 754,
                                "mPoi" : "_9l",
                                "setx" : null,
                                "vo" : [ 2587, 7670, 6140 ]
                             }
                          ],
                          "mPoi" : [
                             [ 3887, 2125, 6097 ],
                             [ 6496, 8560, 9414 ],
                             [ 8391, 2887, 8267 ]
                          ],
                          "pf" : null
                       }
                ],
                "return" : 10
   }
}
```

5.2 C++ GoogleTest code

Each driver class corresponds to a gtest calling class to perform the desired comparison, as shown in below:

```
#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() { delete driverRectangle; }
  DriverRectangle *driverRectangle;
};
```

The comparison of the gtest function is shown in below :

```
TEST_F(GtestRectangle, DriverRectangleGetVolume0) {
  const char *jsonFilePath = "drivervalue/Rectangle/GetVolume0.json";
  Json::Value Root;
  Json::Reader _reader;
  ifstream _ifs(jsonFilePath);
  _reader.parse(_ifs, Root);
  for (int i = 0; i < RECTANGLE_GETVOLUME0_TIMES; i++) {
        driverRectangle->DriverRectangleGetVolume0(i);
        Json::Value GetVolume0_Root = Root["GetVolume0" + to_string(i)];
        /* return */
        int _return_actual = GetVolume0_Root["return"].asInt();
        /* expectreturn */
        int _expectreturn_expected = GetVolume0_Root["expect"].asInt();
        /*  */
        EXPECT_EQ(_return_actual, _expectreturn_expected);
  }
}
```

CHAPTER 6

Parameter capture code generation

6.1 C language parameter capture

The wings parameter capture function is mainly to obtain the function parameter value, global variable value and return value information at runtime.

6.1.1 Naming rules for parameter capture code

The parameter capture code for wings is stored in the paramcaputrecode folder. The naming rules are the same as the driver format, and all driver can be replaced with param.

6.1.2 Example of parameter capture code

As shown in below, for the function coordinate_destroy , information about capture parameters, global variables, and return values is automatically generated.

```
#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);
```

(continues on next page)

(continued from previous page)

```
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
```

Figure below will show the automatically generated code for capturing parameters.

```
int coordinate_destroyTimes = -1;
void ParamCapture_coordinate_destroy(location *loc, size_t length, char *ch)
{
        coordinate_destroyTimes++;
        const char *jsonFile = "paramcapturevalue/wings_c_dem_coordinates/coordinate_
↪destroy.json";
        cJSON *root = NULL;
        if (coordinate_destroyTimes == 0){
                root = cJSON_CreateObject();
        }else{
                char *jsonData = ParamCaptureGetJsonData(jsonFile);
                root = cJSON_Parse(jsonData);
        }
        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", coordinate_destroyTimes);
        cJSON *item = cJSON_CreateObject();
        cJSON_AddItemToObject(root, coordinate_destroy_sp, item);
        free(coordinate_destroy_sp);
        /*It is the 1 parameter: loc */
        cJSON *location_sitem = cJSON_CreateArray();
        Struct_location_s_P(location_sitem, loc, 1);
        cJSON_AddItemToObject(item, "loc", location_sitem);
        /*It is the 2 parameter: length*/
        cJSON_AddItemToObject(item, "length", cJSON_CreateNumber(length));
        /*It is the 3 parameter: ch */
        cJSON_AddItemToObject(item, "ch", cJSON_CreateString(ch));
        FILE *fp;
        fp = fopen(jsonFile, "w");
        fprintf(fp, "%s\n", cJSON_Print(root));
```

(continues on next page)

(continued from previous page)

```
        fclose(fp);
}
```

6.2 C ++ parameter capture

6.2.1 C ++ parameter capture code naming rules

The naming rules for parameter capture of c ++ are the same as that of the driver, just replace the driver with paramcaputre .

6.2.2 C ++ parameter capture code description

For each class, a parameter capture class is generated, and each function in the capture class will generate a function that captures parameters, global variables, and return values.

```
#pragma once
#include "paramcapture.h"
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();
};
```

How to capture a member variable of a class, a capture function is inserted in the class to obtain the private member variable of the class, as shown in Figure 20 :

```
class Rectangle
{
private:
        Point *m_point;
```

(continues on next page)

(continued from previous page)

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

CHAPTER 7

Handling of special assignment types

When dealing with different types, you will encounter some special types of operations, such as void * , function pointers, system types and other special types. For these special types, wings provide different operations for special processing.

7.1 Function parameter is void * and function pointer

For the types of function parameters of void * and function pointers, wings first uses static analysis technology to obtain the specific assignment types when the function parameters are void * and function pointers. For example, the function shown in Figure 21 below:

```
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()
{
```

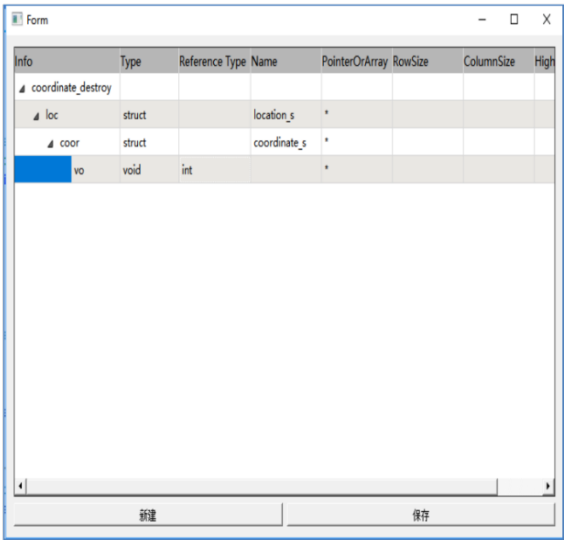
(continues on next page)

(continued from previous page)

```
 func(fun);
}
```

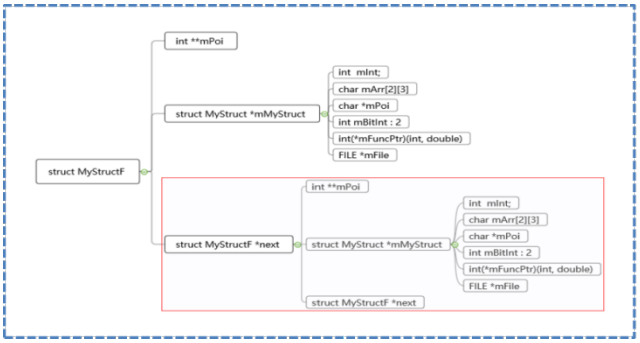
Through static analysis, Wings analyzed that the assignment type of func at the called place was char *. During the assignment process, the parameter of func was assigned to char *. Wings mark the specific assignment type of void * on the data table interface.For some types that cannot be determined by analysis technology, wings will display all relevant functions on the interface, the user will select the required type, and the driver will process the corresponding type. As shown in Figure 22 below:

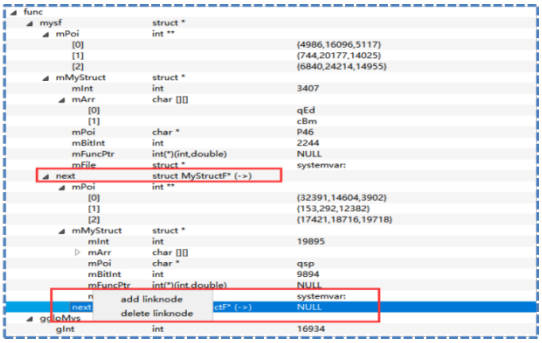
```
#pragma once
#include<stdio.h>
extern int count;
typedef struct coordinate_s{
    /* Pointers to method functions */
    void(*setx)(double, int);
    /* Data */
    int mInt;
    char *mPoi;
    int mArr[2][3];
    void *vo;
} coordinate;

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);
```



7.2 Structure linked list

For the linked list type, a more flexible assignment method is adopted. Taking into account some factors in actual application, we have a two-layer structure for default assignment of the linked list type. In the actual testing process, users can automatically add nodes as needed.





7.3 Structures and classes are system variables

Wings do special template processing for the header files of some C ++ standard libraries and some third-party libraries. In Table 4 below, some special system variables are defined.

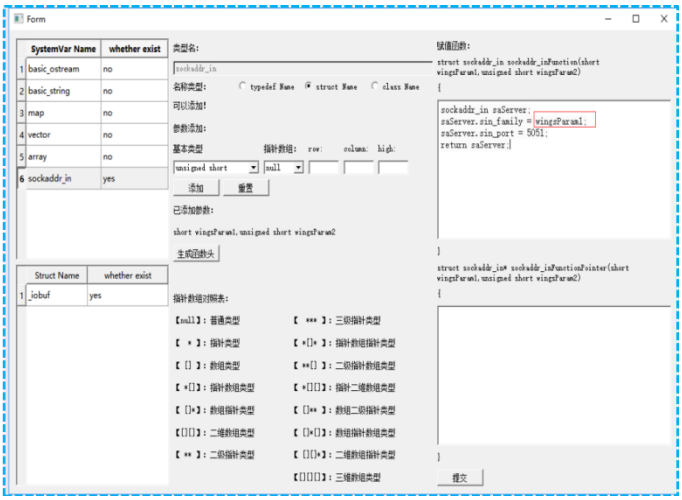
C	C++
The complex type (FILE) used in the C language standard library	STL (std::array、std::forward_list、std::vector Class types in the C ++ standard library (iostream) STL (std :: array , std :: forward_list , std :: vector)
Third-party libraries (cJSON)	Third-party libraries called by C ++
Complex types where users need special assignments	Class types where users need special assignments

- When encountering a special type of assignment, such as sockaddr_in in the following figure, first of all, this type is marked as a type in the system header file and requires special treatment.
- Member variables in sockaddr_in require user-defined assignment
- First, wings detects that sockaddr_in needs special treatment, and it will display this variable in the interface of the template class.
- Then for the variables that need special treatment, such as sin_family, etc., the user needs to configure for the required type.
- After completing configuration, returns the corresponding sockaddr_in structural body to the object.
- Driver code is generated, the function written by the user will be called, and the variables that need to be filled, such as the sin_family value, are automatically generated by us.

Wings are for special templates and have a simple interface for configuration processing.

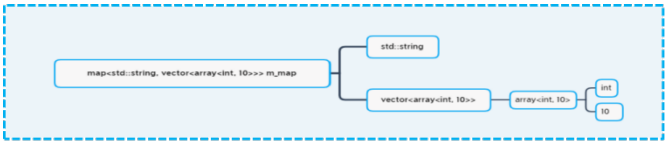
```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

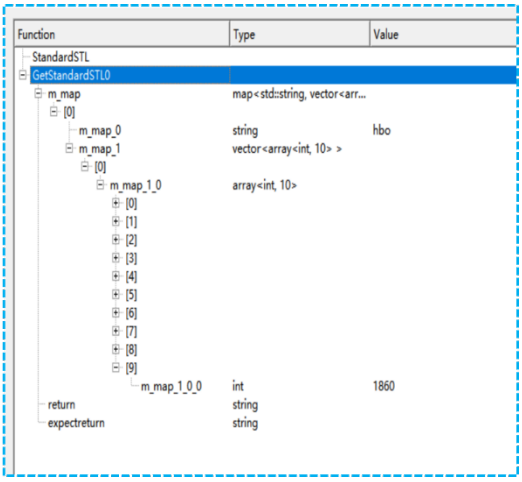
```
#pragma once
#include <iostream>
#include <winsock.h>
#pragma comment(lib,"ws2_32.lib")
class SocketDemo
{
    void creatSocket(struct sockaddr_in saServer);

};
```



7.4 stl standard template library

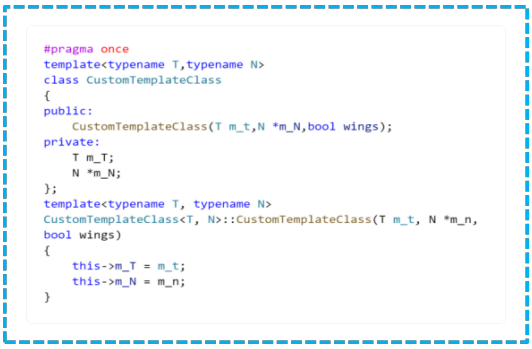
For the standard container in C ++ , we expand and assign the types in the container, and generate a set of values by default. You can click on the interface to add according to your needs.





7.5 C ++ custom template class

For the template type whose parameters are customized, we will analyze the specific type of the template class. For example, the type of the template class in GetTest in Figure 29 is int and double , and the type in GetTestDemo is std :: string and double . Insert a constructor CustomTemplateClass , when actually constructing the template class object, call the specific assignment type to construct.



```
#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;
}
```

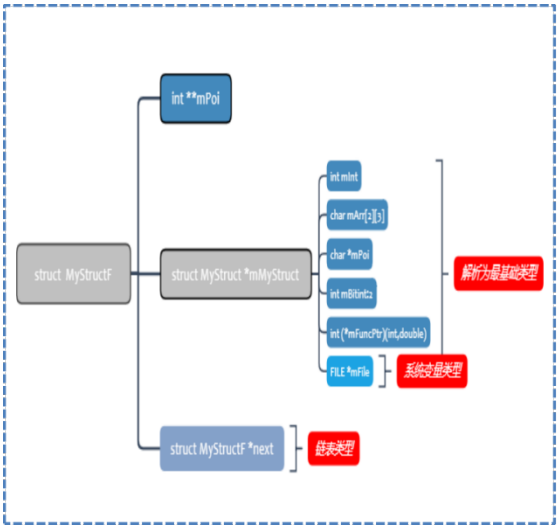
```
#pragma once
#include"CustomTemplateClass.hpp"
#include<string>
class TemplateClassTest
{
public:
    TemplateClassTest();
    ~TemplateClassTest();
    void GetTest(CustomTemplateClass<int, double> cum);
    void GetTestDemo(CustomTemplateClass<std::string, double> cum1
);
private:

};
```

CHAPTER 8

Data table

Wings test case data is randomly generated and supports int, char, double, float, bool, char * types. The data table supports editing any value.

1. The wings data table expands layer by layer to basic types for parameters.
2. Pointer types for basic types, such as int * p; wings are processed as one-dimensional array types of indefinite length, int ** p; are processed as two-dimensional array types of indefinite length, the default length is 3 , and the data table interface can be Click to add and delete data.
3. Arrays of indefinite length are used as function parameters, such as int p []; the default length of wings is 1 , and users can add and modify them arbitrarily on the data table interface according to needs.



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