

## Testing C with **Cantata++**



Certificate Number FM1589

Last Updated: 04/10/2001

Copyright © **IPL** Information Processing Ltd 2001.

This document is the exclusive copyright of IPL Information Processing Limited (IPL). If you are a Customer, as defined in a current Software Licence Agreement with IPL or one of its authorised suppliers, you may view and print this document solely to assist in your use of the AdaTEST, Cantata and Cantata++ products. If you are not a Customer, as defined in a current Software Licence Agreement with IPL or one of its authorised suppliers, you have no rights to view or print this document. For the avoidance of doubt you may not otherwise copy, store, retransmit, publish, change, deface or reproduce this document in electronic, hard-copy or any other form without IPL's prior written consent.

## **Contents**

<b>1</b>	<b><i>Introduction</i></b> .....	<b>4</b>
1.1	Scope .....	4
<b>2</b>	<b><i>Differences between Cantata++ for C and C++</i></b> .....	<b>5</b>
2.1	CHECK functions .....	5
2.1.1	CHECK_<TYPE> .....	5
2.1.2	CHECK_<TYPE>_RANGE .....	6
2.1.3	CHECK_MEMORY & CHECK_MEMORY_WARN .....	6
2.1.4	CHECK_OBSERVATION & CHECK_OBSERVATION_WARN .....	7
2.2	Other Test Harness Directives NOT Supported in C mode .....	7
2.3	A Template Cantata++ C Test Script .....	7
<b>3</b>	<b><i>Convert Example</i></b> .....	<b>9</b>
3.1	Testing the Convert Example .....	10

## **1 Introduction**

Cantata++ is a complete testing solution for both C and C++. This technical note discusses the mechanisms that should be employed in testing C with Cantata++. Cantata++ for C has a number of significant improvements over Cantata, IPL's original C testing tool. Cantata++ uses a state of the art front-end parser for all its parse tools, which means that it can cope much better with third party code, such as compiler headers, that do not exactly adhere to the ANSI C standard. Cantata++ also provides wrapping, something that is not offered by any other tool in this domain, and an improved coverage analyzer.

### **1.1 Scope**

This technical note is relevant to anyone wishing to test C only projects with Cantata++, i.e. where all the source code and test scripts are written as ANSI C.

## 2 Differences between Cantata++ for C and C++

Cantata++ for C will be used to test C modules, therefore we will be compiling the projects as C and so the test script should also be written in C. Even if it is possible to have both C and C++ units in your project, mixed mode compilation often causes problems and is best avoided. If we are to limit our projects, and thus our test scripts, to C only then there a number of C++ test harness directives that cannot be used. This section lists the test harness directives that are C++ only and provides a description of the C alternatives. For more information please refer to the [Cantata++ Test Harness manual](#).

### 2.1 CHECK functions

Templates are not supported in C so the Cantata++ template CHECK functions cannot be used in a C test script. The following CHECK directives are template functions and therefore are NOT available in Cantata++ for C:

- a) CHECK()
- b) CHECK\_WARN()
- c) CHECK\_NAMED()
- d) CHECK\_NAMED\_WARN()
- e) CHECK\_RANGE()
- f) CHECK\_RANGE\_WARN()
- g) CHECK\_RANGE\_NAMED()
- h) CHECK\_RANGE\_NAMED\_WARN()
- i) CHECK\_DYNAMIC()
- j) CHECK\_DYNAMIC\_NAMED()

Instead of the above template functions the specific CHECK\_<TYPE> functions can be used. The following CHECK directives are available in Cantata++ for C.

#### 2.1.1 CHECK\_<TYPE>

```
int CHECK_ADDRESS(const char* text,
                 const void* actual,
                 const void* expected);

int CHECK_BOOLEAN(const char* text,
                 int actual,
                 int expected);

int CHECK_CHAR(const char* text,
              char actual,
              char expected);

int CHECK_DOUBLE(const char* text,
                long double actual, /* double for K&R C */
                long double expected); /* double for K&R C */

int CHECK_S_CHAR(const char* text,
                signed char actual,
                signed char expected);

int CHECK_S_INT(const char* text,
               signed long actual, /* or signed long long */
               signed long expected); /* or signed long long */

int CHECK_STRING(const char* text,
                const char* actual,
                const char* expected);
```

```

int CHECK_U_CHAR(const char* text,
                unsigned char actual,
                unsigned char expected);

int CHECK_U_INT(const char* text,
               unsigned long actual, /* or unsigned long long */
               unsigned long expected); /* or unsigned long long */

int CHECK_WCHAR(const char* text,
               wchar_t actual,
               wchar_t expected);

int CHECK_WSTRING(const char* text,
                 const wchar_t* actual,
                 const wchar_t* expected);

```

## 2.1.2 CHECK\_<TYPE>\_RANGE

```

int CHECK_ADDRESS_RANGE(const char* text,
                      const void* actual,
                      const void* lower,
                      const void* upper);

int CHECK_CHAR_RANGE(const char* text,
                   char actual,
                   char lower,
                   char upper);

int CHECK_DOUBLE_RANGE(const char* text,
                     long double actual, /* double for K&R C */
                     long double lower, /* double for K&R C */
                     long double upper); /* double for K&R C */

int CHECK_S_CHAR_RANGE(const char* text,
                     signed char actual,
                     signed char lower,
                     signed char upper);

int CHECK_S_INT_RANGE(const char* text,
                    signed long actual, /* or signed long long */
                    signed long lower, /* or signed long long */
                    signed long upper); /* or signed long long */

int CHECK_STRING_RANGE(const char* text,
                     const char* actual,
                     const char* lower,
                     const char* upper);

int CHECK_U_CHAR_RANGE(const char* text,
                     unsigned char actual,
                     unsigned char lower,
                     unsigned char upper);

int CHECK_U_INT_RANGE(const char* text,
                    unsigned long actual, /* or unsigned long long */
                    unsigned long lower, /* or unsigned long long */
                    unsigned long upper); /* or unsigned long long */

int CHECK_WCHAR_RANGE(const char* text,
                    wchar_t actual,
                    wchar_t lower,
                    wchar_t upper);

int CHECK_WSTRING_RANGE(const char* text,
                      const wchar_t* actual,
                      const wchar_t* lower,
                      const wchar_t* upper);

```

## 2.1.3 CHECK\_MEMORY & CHECK\_MEMORY\_WARN

```

bool CHECK_MEMORY(const char* text,

```

```

        const void* actual,
        const void* expected,
        size_t size);

bool CHECK_MEMORY_WARN(const char* text,
        const void* actual,
        const void* expected,
        size_t size);

```

### 2.1.4 CHECK\_OBSEVATIION & CHECK\_OBSERVATION\_WARN

```

bool CHECK_OBSERVATION(const char* text,
        char response = 'Y');

bool CHECK_OBSERVATION_WARN(const char* text,
        char response = 'Y');

```

## 2.2 Other Test Harness Directives NOT Supported in C mode

The following Test Harness directives are also NOT supported in Cantata++ for C:

- k) TEST\_CLASS(TYPE)
- l) FACTORY(TYPE)
- m) OUTPUT\_VALUE()
- n) DECLARE\_EXPECTED()
- o) DECLARE\_EXPECTED\_NAMED()
- p) EXPECTED()
- q) CHECK\_ALL()
- r) START\_EXCEPTION
- s) END\_EXCEPTION
- t) NO\_EXCEPTIONS
- u) EXPECT\_EXCEPTION()
- v) OR\_EXCEPTION()

All 'Call Sequence Validation Directives', 'Timing Analysis Directives', 'Log File Handling Directives' and 'Low-Level API Directives' are supported in C mode.

## 2.3 A Template Cantata++ C Test Script

The following is an example Cantata++ C test script:

```

/*****
/*
/*      Copyright (C) 2001; IPL, Information Processing Ltd
/*
/*
/*****
/*      <c_template.c>:  Template Test Script for C
/*
/*
/*****

#include <cppth.h>    /* C Test Harness directives    */
#include <cppcg.h>    /* Coverage Gathering directives */
#include <cppcr.h>    /* Coverage Reporting directives */

/* Insert required #include directives here */

void test_1()
{
    START_TEST("Test name here", "Description of test here "
              "and here");

    /* Expected Calls sequence. e.g. CPPTH_EXPECTED_CALLS */

```

```

        /* Set up input data here */

        /* Run software under test here */

        /* Check output data here */

        /* End expected calls. e.g. CPPTH_END_CALLS */

    END_TEST();
}

void test_2()
{
    START_TEST("Test name here", "Description of test here "
              "and here");

    /* Test case here */

    END_TEST();
}

void test_3()
{
    START_TEST("Test name here", "Description of test here "
              "and here");

    /* Test case here */

    END_TEST();
}

void run_test()
{
    /*
    * Put DEFAULT_EXPECTED_CALLS here
    * e.g. DEFAULT_EXPECTED_CALLS("{optional_func1;optional_func2}");
    */

    /* Call test cases */
    test_1();
    test_2();
    test_3();

    /*
    * Add analysis directives here
    *
    * Define the software under test
    * #define CPPCA_SUT "*"
    * Include the coverage directives
    * #include "analysis_file.anl"
    */
}

int main()
{
    /*
    * Include IDE Enhanced Output directives here
    * e.g. CONFIGURE_ENHANCED_OUTPUT(cppth_enhanced_op_stdout |
    *                               cppth_enhanced_op_fail |
    *                               cppth_enhanced_op_covg);
    */

    START_SCRIPT("test script name", 3);

    run_test();

    END_SCRIPT();
    return 0;
}

/*****/

```

### 3 Convert Example

`convert` is a simple C function to convert a decimal integer into either a decimal, hexadecimal or octal string. The function takes 3 parameters, one specifying the required base for the output string, one specifying the input value and one output parameter that will contain the converted value as a string. `convert` also returns an enumerated type, `retcodes`, representing the level of success. The prototype for `convert` is defined as:

```
extern enum retcodes convert ( enum bases type, int number, char *output );
```

The `retcodes` and `bases` enumerated types are defined as follows:

```
enum bases { DEC=0, OCT=1, HEX=2 };
enum retcodes { OK=0, RANGE=1, ILLEGALOP=2 };
```

The `convert` function is defined in `convert.c` as follows:

```

/*****
/*
/* Convert Function
/*
/* Converts integer to string format in DECIMAL, OCTAL or HEX
/*
/* File: convert.c
/*
/*
/*****

/*
 * Include Header files
 */
#include <string.h>
#include "convert.h"
#include "range_ok.h"

/*
 * CONVERT module source
 */
enum retcodes convert (enum bases type, int number, char *output)
{
    enum retcodes ret_val=OK;
    int no_of_digits;
    int base;
    int digit;
    int temp_number;
    char string[16]={ "          " };

/*
 * If number is outside valid range, return error code of RANGE
 */
    ret_val=range_ok(number);
    if (ret_val == OK)
    {

/*
 * Determine type of conversion to be performed
 */
        switch (type)
        {
            case OCT:
                base=8;
                break;

            case DEC:
                base=10;
                break;

            case HEX:
                base=16;
                break;
        }
    }
}

```

```

/*
 * Default case - Illegal conversion type specified so return error ILLEGALOP
 */
    default:
        ret_val=ILLEGALOP;
        break;
    }
/*
 * If legal conversion type was specified, do conversion
 */
    if (ret_val==OK)
    {
        temp_number = number;

        no_of_digits = 1;
        do
        {
            digit = temp_number;
            temp_number /= base;
            digit = digit - ( temp_number * base );
        }
/*
 * Check whether Hex non-numeric character wanted
 */
        if (digit > 9)
        {
            digit += ('A' - 10);
        }
        else
        {
            digit += '0';
        }
        string [15-no_of_digits] = (char) digit;
        no_of_digits++;
    }
    while (temp_number != 0);
/*
 * Now output string
 */
    (void) strcpy( output, (string + (16 - no_of_digits)) );
}
/*
 * Return error_code
 */
return ret_val;
}

```

`convert` makes one external call to a function called `range_ok`. The `range_ok` function takes just one parameter, an integer value, which it then checks against the minimum and maximum accepted values. The return is a retcodes enumerated type value specifying whether the value was in range. The prototype for `range_ok` is as follows:

```
extern enum retcodes range_ok ( int number );
```

### 3.1 Testing the Convert Example

The first task is to determine the different possible test that you can use to exercise the `convert` function. For example:

- a) Decimal conversion of an in-range value
- b) Hexadecimal conversion of an in-range value
- c) Octal conversion of an in-range value
- d) Conversion of an out-of-range value
- e) Force illegal operation error
- f) Conversion of a hexadecimal non-numeric value

Each of the test cases initialises the two input parameters ('type' and 'number') and must check both the output parameter ('output') and the function return value. Coverage Analysis checks can also be added. Start with a template test script such as that given in [section 2.3](#) and test cases one-by-one. An example completed test script is given below:

```

/*****
/*
/*      Copyright (C) 2001; IPL, Information Processing Ltd
/*
/*
/*****
/*
/*      test_convert.c:  Convert Test Script
/*
/*
/*****

#include <cppth.h>    /* Test Harness directives      */
#include <cppcg.h>    /* Coverage Gathering directives */
#include <cppcr.h>    /* Coverage Reporting directives */
#include "convert.h"
#include "range_ok.h"

/* Prototypes for test functions */
void run_test(void);
void test_1(void);
void test_2(void);
void test_3(void);
void test_4(void);
void test_5(void);
void test_6(void);

/* Global test data */
char output[10];      /* convert output string */
enum retcodes R_convert; /* convert return value */

/*****
/* Test cases
/*****

void run_test()
{
    test_1();
    test_2();
    test_3();
    test_4();
    test_5();
    test_6();

    /* Define the software under test */
    #define CPPCA_SUT "convert"
    /* Include the coverage directives */
    #include "convert.anl"
}

void test_1()
{
    START_TEST("Decimal 100", "Decimal conversion of in-range value");

    EXPECTED_CALLS("1*range_ok#100_OK");

    /* Set up input data here */
    strcpy(output, "0x55555555");
    R_convert = 0x55555555;

    /* Run software under test here */
    R_convert = convert(DEC, 100, output);

    /* Check outputs */
    CHECK_STRING(output, "100");
    CHECK_S_INT(R_convert, OK);

    END_CALLS();
}

```

```
    END_TEST();
}

void test_2()
{
    START_TEST("Hex 100", "Hexadecimal conversion of in-range value");

    EXPECTED_CALLS("1*range_ok#100_OK");

    /* Set up input data here */
    strcpy(output, "0x55555555");
    R_convert = 0x55555555;

    /* Run software under test here */
    R_convert = convert(HEX, 100, output);

    /* Check outputs */
    CHECK_STRING(output, "64");
    CHECK_S_INT(R_convert, OK);

    END_CALLS();
    END_TEST();
}

void test_3()
{
    START_TEST("Octal 100", "Octal conversion of in-range value");

    EXPECTED_CALLS("1*range_ok#100_OK");

    /* Set up input data here */
    strcpy(output, "0x55555555");
    R_convert = 0x55555555;

    /* Run software under test here */
    R_convert = convert(OCT, 100, output);

    /* Check outputs */
    CHECK_STRING(output, "144");
    CHECK_S_INT(R_convert, OK);

    END_CALLS();
    END_TEST();
}

void test_4()
{
    START_TEST("Range error", "Conversion on out of range value");

    EXPECTED_CALLS("1*range_ok#32333_RANGE");

    /* Set up input data here */
    strcpy(output, "0x55555555");
    R_convert = 0x55555555;

    /* Run software under test here */
    R_convert = convert(DEC, 32333, output);

    /* Check outputs */
    /* range error = output undefined */
    CHECK_S_INT(R_convert, RANGE);

    END_CALLS();
    END_TEST();
}

void test_5()
{
    START_TEST("Illegal operation", "Illegal conversion type specified");

    EXPECTED_CALLS("1*range_ok#100_OK");

    /* Set up input data here */
    strcpy(output, "0x55555555");
    R_convert = 0x55555555;

    /* Run software under test here */
```

```

        R_convert = convert(33, 100, output);

        /* Check outputs */
        /* operation error = output undefined */
        CHECK_S_INT(R_convert, ILLEGALOP);

        END_CALLS();
    END_TEST();
}

void test_6()
{
    START_TEST("Hex non-numeric", "Hexadecimal conversion to non-numeric
character");

    EXPECTED_CALLS("1*range_ok#10_OK");

    /* Set up input data here */
    strcpy(output, "0x55555555");
    R_convert = 0x55555555;

    /* Run software under test here */
    R_convert = convert(HEX, 10, output);

    /* Check outputs */
    CHECK_STRING(output, "A");
    CHECK_S_INT(R_convert, OK);

    END_CALLS();
    END_TEST();
}

int main()
{
    CONFIGURE_ENHANCED_OUTPUT(cppth_enhanced_op_stdout |
                              cppth_enhanced_op_fail |
                              cppth_enhanced_op_covg);

    OPEN_LOG("convert.ctr", false);
    SET_LOG_LEVEL(cppth_ll_detailed);
    START_SCRIPT("convert", true);

    run_test();

    return !END_SCRIPT(true);
}

/*****
/* Stub Functions */
*****/

/* range_ok stub */
extern enum retcodes range_ok(int number)
{
    /* Register call */
    const char *call_id = CPPTH_REG_CALLFUNC("range_ok");
    (void)CPPTH_REG_CALLINST(call_id);

    // Actions
    IF_INSTANCE("100_OK")
    {
        CHECK_S_INT(number, 100);
        return OK;
    }
    IF_INSTANCE("32333_RANGE")
    {
        CHECK_S_INT(number, 32333);
        return RANGE;
    }
    IF_INSTANCE("10_OK")
    {
        CHECK_S_INT(number, 10);
        return OK;
    }

    /* Illegal stub instance */
    return *(enum retcodes *)0;
}

```

```
}  
/*****/
```