Unit testing

Dr. Yihsiang Liow (January 4, 2014)

Contents

1 General information

A unit test generally tests the smallest unit of a software system. That means either a function or a method (or even a class).

```
File:  assert/assert.tex
```

2 assert

Try this:

```
#include <iostream>
#include <cassert>

int main()
{
    assert(0);
    std::cout << "hello world" << '\n';
    return 0;
}
```

Compile and run the program:

```
[student@localhost tmp] g++ *.cpp; ./a.out
a.out: main.cpp:6: int main(): Assertion `0' failed.
/bin/sh: line 1: 19472 Aborted                 (core dumped) ./a.out
```

Note that an assertion message is printed, telling you that the program was aborted. Note that the print statement in the program was not executed.

Calling `assert()` with `0` will cause your program to abort. `assert()` is frequently used to check for conditions when developing a program. For instance suppose an input to a function `f()` should be an integer greater than 0. You can do this:

```
void f(int x)
{
    assert(x > 0);
    // ... do something with x ...
}
```

Or suppose the function `f()` is suppose to modify `x`, which is passed by reference, so that on return, `x` must be less than 10. You can do this:

```
void f(int & x)
{
    assert(x > 0);
    // ... do something to x ...
    assert(x < 10);
}
```

The conditions guarding the entrance into the main code of `f()` and the exit from `f()` are frequently called precondition(s) and postcondition(s).

Note that the failure of the conditions (pre- or post-) should be viewed as a program error. In other words, if the system is developed correctly, you do expect the pre- and postconditions to be satisfied.

As you can see from the example, you are using C++ type checking to ensure that the paramter `x` is an integer *and* you are using the precondition to ensure that `x` is positive. In a sense, you can trying to simulate the existence of a positive integer type, except that it's not exactly a type but a runtime condition. Therefore if the condition fails, it can only fail during runtime and not during compile time.

But `assert()` is not just for tightening your control over types. Besides using `assert()` for a condition on a single variable, you can also have conditions that involves more than one. For instance

```
void area_right_angle_triangle(double a, double b, double h)
{
    assert(a > 0);
    assert(b > 0);
    assert(h > 0);
    assert(a * a + b * b == h * h);
    // ... return the area ...
}
```

The parameters `a`, `b`, and `h` represents the sides of a right-angle triangle and therefore must satisfy the Pythagorean condition. In this case the last condition involves 3 variables and has nothing to do with types. By the way, since doubles are not exact, you probably want to do this (or something similar):

```
void area_right_angle_triangle(double a, double b, double h)
{
    assert(a > 0);
    assert(b > 0);
    assert(h > 0);
    assert(fabs(a * a + b * b - h * h) < 0.00001);
    // ... return the area ...
}
```

and you can further combine all the conditions together into one if you wish:

```
void area_right_angle_triangle(double a, double b, double h)
{
    assert(a > 0 && b > 0 && h > 0
           && fabs(a * a + b * b - h * h) < 0.00001);
    // ... return the area ...
}
```

In a “serious” software system (say a critical system), pre- and postconditions are fully documented (yes, that’s right). In that case you should use one assert per condition and label it of do something like this:

```
double area_right_angle_triangle(double a, double b, double h)
{
    assert(a > 0                                      // Precondition 23
           && b > 0                                   // Precondition 24
           && h > 0                                   // Precondition 25
           && fabs(a * a + b * b - h * h) < 0.00001); // Precondition 26
    // ... return the area ...
}
```

2.1 Maintenance and performance benefit

Now suppose you have checked your software with lots of test cases and your program pass all assert tests. At that point, you feel that you can speed up your program a little and not perform the checks. (The execution of `assert()` of course takes time.) You know that the assert checks might be usedful in the future. You have decided to keep them all the asserts and do this:

```
double area_right_angle_triangle(double a, double b, double h)
{
    //assert(a > 0                                     // Precondition 23
    //       && b > 0                                  // Precondition 24
    //       && h > 0                                  // Precondition 25
    //       && fabs(a * a + b * b - h * h) < 0.00001); // Precondition 26
    // ... return the area ...
}
```

Right? No! All you need to do is to realize that you can actually turn off asserts by defining the "no debug" preprocessor macro, `NDEBUG`. Compile and run this:

```
#include <iostream>
#define NDEBUG
#include <cassert>

int main()
{
    assert(0);
    std::cout << "hello world" << '\n';
    return 0;
}
```

Compile and run the program:

```
[student@localhost tmp] g++ *.cpp; ./a.out
hello world
```

Notice there is no assertion message. Note that the `#define NDEBUG` must appear before `#include <cassert>`. Another way to achieve the same thing is to use the "no debug" option when you execute g++:

```
#include <iostream>
#include <cassert>

int main()
{
    assert(0);
    std::cout << "hello world" << '\n';
    return 0;
}
```

Compile and run the program:

```
[student@localhost tmp] g++ *.cpp -DNDEBUG; ./a.out
hello world
```

Notice again that the assertion message is not printed.

2.2 Modifying assert

When you think of assertion catching condition failure, you would recall exceptions right away. In the case of an exception, you have an opportunity to catch it and do something with it, such as printing some error message or you might want to log an error message (example: save it to a file). In the case of assertions, you cannot catch them. For instance say you have this:

```
void f(int x, int y)
{
    assert(x < y);
    // ...
}
```

and the condition fails. It would nice to see a print out of the value of `x` and the value of `y`. You can of course go back to exceptions:

```
void f(int x, int y)
{
    try
    {
        if (!(x < y)) throw Exception("%s < %s condition fails")
        // ...
    }
    except (Exception & e)
    {
        std::cout << e << std::endl; // or use std::err
        assert(0);
    }
}
```

Wow! What a pain!

Another way to do this is to write your own `assert()`. Here's one possibility:

```
#include <sstream>

std::string int_to_string(int i)
{
    std::ostringstream cout;
    cout << i;
    return cout.str();
}

void myassert(bool b, std::string & s)
{
    if (!b) std::cout << s << std::endl; // or use std::err
    assert(b);
}

void f(int x, int y)
{
    myassert(x < y, "x < y fails for x = ");
    // ...
}
```

http://stackoverflow.com/questions/2193544/how-to-print-additional-information-when-a

Tutorial on CppUnit.

Installation (f18):

```
yum -y install cppunit
```

File: typeinfo/typeinfo.tex

3 typeinfo

typeinfo is helpful for introspection.

```
#include <iostream>
#include <typeinfo>

int main()
{
    int x = 42;
    std::cout << (typeid(x) == typeid(int)) << '\n';
    std::cout << (typeid(42) == typeid(int)) << '\n';
    std::cout << typeid(int).name() << '\n';
    std::cout << typeid(unsigned int).name() << '\n';
    std::cout << typeid(char).name() << '\n';
    std::cout << typeid(bool).name() << '\n';
    std::cout << typeid(int *).name() << '\n';
    std::cout << typeid(int []).name() << '\n';
    std::cout << typeid(int &).name() << '\n';
    std::cout << (typeid(int &) == typeid(int)) << '\n';

    return 0;
}
```

```
[student@localhost typeinfo] g++ main.cpp; ./a.out
1
1
i
j
c
b
Pi
A_i
i
1
```

Note that the `int` and `int &` have the same type id.