# Testing ADTs in C++

Steven J Zeil

February 13, 2013

## Contents

It would be nice if every new ADT we wrote worked correctly the first time it compiled properly, but the real world just doesn't work that way.

One advantage of organizing your code into a collection of ADTs is that ADTs provide, not only a convenient way to package up the functionality of your code, but also a convenient basis for testing.

The *Unit test frameworks give us a powerful tool for unit testing. But we still need to pick the tests.

# 1 Be Smart: Generate and Check

**Testing addContact**

In our earlier test for addContact, we weren't particularly thorough:

```
TEST_F (MailingListTests, addContact) {
```



```
  mlist.addContact (jones);
  EXPECT_TRUE (mlist.contains("Jones"));
  EXPECT_EQ ("Jones", mlist.getContact("Jones").getName());
  EXPECT_EQ (4, ml.size());
}
```

- Missing functional case: adding a contact that already exists
- Missing boundary/special case: adding to an empty container
- Missing special cases: Adding to beginning or end of an ordered sequence

. . . . . .




**Adding Variety**

Some of our concerns could be addressed by adding tests but varying the parameters:

```
TEST_F (MailingListTests, addExistingContact) {
  mlist.addContact (baker);
  EXPECT_TRUE (mlist.contains("Baker"));
  EXPECT_EQ ("Baker", mlist.getContact("Baker").getName());
  EXPECT_EQ (3, ml.size());
}
```

. . . . . .

**Generate and Check**

A useful design pattern for producing well-varied tests is

```
for v: varieties of ADT {
```



```
   ADT x = generate(v);
   result = applyFunctionBeingTested(x);
   check (x, result);
}
```

- `generate` and `check` could be separate functions or in-line
  - depends on whether you can re-use in multiple tests

.....

**Generate and Check**

```
for v: varieties of ADT {
   ADT x = generate(v);
```




```
   result = applyFunctionBeingTested(x);
   check (x, result);
}
```

- Most common "variety" would be size
  - Could also be different constructors (in which case you might not literally have a loop:

```
ADT x (constructor1);
result = applyFunctionBeingTested(x);
check (x, result);

ADT x2 (constructor2, ...);
result = applyFunctionBeingTested(x2);
```




```
            check (x2, result);

                        ..............................
```

**Example: addContact**

A more elaborate fixture will aid in generating mailing lists of different sizes:

```
// The fixture for testing class MailingList.
class MailingListTests  {
public:
  Contact jones;
  vector<Contact> contacts;

  MailingListTests() {
```




```
  jones = Contact("Jones", "21 Penn. Ave.");
  contacts.clear();
  contacts.push_back (Contact ("Baker", "Drury Ln."));
  contacts.push_back (Contact ("Holmes", "221B Baker St."));
  contacts.push_back(jones);
  contacts.push_back (Contact ("Wolfe", "454 W. 35th St."));
}

~MailingListTests() {
}


MailingList generate(int n) const
{
```




```
    MailingList m;
    for (int i = 0; i < n; ++i)
      m.addContact(contacts[i]);
    return m;
  }

};
```

.....

**Example: addContact - many sizes**

```
BOOST_FIXTURE_TEST_CASE (addContact, MailingListTests) {
  for (unsigned sz = 0; sz < contacts.size(); ++sz)
```




```
    {
      MailingList ml0 = generate(sz);
      MailingList ml (ml0);
      bool alreadyContained = ml.contains("Jones");
      ml.addContact (jones);
      BOOST_CHECK (ml.contains("Jones"));
      BOOST_CHECK_EQUAL ("Jones", ml.getContact("Jones").getName());
      if (alreadyContained)
      BOOST_CHECK_EQUAL (ml.size(), sz);
      else
    BOOST_CHECK_EQUAL (ml.size(), sz+1);
    }
}
```




- Here we generate mailing lists of a variety of sizes and add Jones to them
- At larger sizes, Jones is already in the list – functional case covered
- `sz == 0` covers one of our boundary/special cases

.....

**Example: addContact - ordering**

```
BOOST_FIXTURE_TEST_CASE (addContact, MailingListTests) {
  for (unsigned sel = 0; sel < contacts.size(); ++sel)
    {
      Contact& toAdd = contacts[sel];
      const Name& nameToAdd = contacts[sel];
```




```
    for (unsigned sz = 0; sz < contacts.size(); ++sz)
      {
        MailingList ml0 = generate(sz);
        MailingList ml (ml0);
        bool alreadyContained = ml.contains(nameToAdd);
        ml.addContact (toAdd);
        BOOST_CHECK (ml.contains(nameToAdd));
        BOOST_CHECK_EQUAL (nameToAdd, ml.getContact(nameToAdd).getName()
        if (alreadyContained)
          BOOST_CHECK_EQUAL (ml.size(), sz);
        else
          BOOST_CHECK_EQUAL (ml.size(), sz+1);
      }
  }
```




```
}
```

- We an also explore adding to different positions (beginning, middle, end)
  - by varying which element we add

. . . . . .

## 2 Be Thorough: Mutators and Accessors

OK, let's say we want to design a unit test suite for our *MailingList* ADT.

Just staring at the ADT interface, where do we start? The criteria we suggested earlier (typical values, extremal values, special values) may be of help, but it's far from obvious how to apply those ideas.



There is an organized way to approach this test design. It starts with the recognition that the member functions of an ADT can usually be divided into two groups - the mutators and the accessors. *Mutator* functions are the functions that alter the value of an object. These include constructors and assignment operators. *Accessor* functions are the functions that "look at" the value of an object but do not change it. Some functions may both alter an object and return part of its value - we'll have to wing it a bit with those.

**Mutators and Accessors**

To test an ADT, divide the public interface into

- *mutators*: functions that alter the value of the object
- *accessors*: functions that "look at" the current value of an object
  - Occasional functions will fall in both classes



**Organizing ADT Tests**

The basic procedure for writing an ADT unit test is to

1. Consider each mutator in turn.
2. Write a test that begins by applying that mutator function.
    - Then consider how that mutator will have affected the results of each accessor.
    - Write assertions to test those effects.

Commonly, each mutator will be tested in a separate function.




## 2.1 Example: Unit Testing of the MailingList Class

**Example: Unit Testing of the MailingList Class**

```
#ifndef MAILINGLIST_H
#define MAILINGLIST_H

#include <iostream>
#include <string>

#include "contact.h"

/**
   A collection of names and addresses
*/
```




```
class MailingList
{
public:
  MailingList();
  MailingList(const MailingList&);
  ~MailingList();

  const MailingList& operator= (const MailingList&);

  // Add a new contact to the list
  void addContact (const Contact& contact);

  // Does the list contain this person?
  bool contains (const Name&) const;
```




```
// Find the contact
const Contact& getContact (const Name& nm) const;
//pre: contains(nm)

// Remove one matching contact
void removeContact (const Contact&);
void removeContact (const Name&);

// combine two mailing lists
void merge (const MailingList& otherList);

// How many contacts in list?
int size() const;
```




```
  bool operator== (const MailingList& right) const;
  bool operator< (const MailingList& right) const;

private:

  struct ML_Node {
    Contact contact;
    ML_Node* next;

    ML_Node (const Contact& c, ML_Node* nxt)
      : contact(c), next(nxt)
    {}
```




```
  };

  ML_Node* first;
  ML_Node* last;
  int theSize;

  // helper functions
  void clear();
  void remove (ML_Node* previous, ML_Node* current);

  friend std::ostream& operator<< (std::ostream& out, const MailingList& a
};

// print list, sorted by Contact
```




```
std::ostream& operator<< (std::ostream& out, const MailingList& list);


#endif
```

Look at our `MailingList` class.
What are the mutators? What are the accessors?

.....

**MailingList Mutators**

The mutators are

- the two constructors,
- the assignment operator,




- `addContact`,
- both `removeContact` functions, and
- `merge`.

.....

**MailingList Accessors**

The accessors are

- `size`,
- `contains`
- `getContact`




- `operator==` and `operator<`
- the output operator `operator«`

. . . . . .

**Unit Test Skeleton**

Here is the basic skeleton for our test suite.

```
#define BOOST_TEST_MODULE MailingList test

#include "mailinglist.h"

#include <string>
#include <sstream>
```




```
#include <vector>

#include <boost/test/included/unit_test.hpp>

using namespace std;
```

Now we start going through the mutators.

. . . . . .

**Testing the Constructors**

The first mutators we listed were the two constructors. Let's start with the simpler of these.

. . . . . .



**Apply The Mutator**

```
BOOST_AUTO_TEST_CASE ( constructor ) {
    MailingList ml;
      ⋮
```

First we start by applying the mutator.

. . . . . .

**Apply the Accessors to the Mutated Object**

Then we go down the list of accessors and ask what we expect each one to return.

```
BOOST_AUTO_TEST_CASE ( constructor ) {
    MailingList ml;
    BOOST_CHECK_EQUAL (0, ml.size());
    BOOST_CHECK (!ml.contains("Jones"));
```



```
    BOOST_CHECK_EQUAL (ml, MailingList());
    BOOST_CHECK (!(ml < MailingList()));
}
```

1. It's pretty clear, for example, that the size() of the list will be 0
2. `contains()` would always return false
3. The EQUAL test checks the `operator==`
4. We check for consistency of `operator<`

We can't check the accessors

- `getContact`




  - can't satisfy the pre-condition, and

- `operator«`
  - behavior unspecified

.....

**Testing the Copy Constructor**

```
BOOST_FIXTURE_TEST_CASE ( copyConstructor,  MailingListTests) {
  for (unsigned sz = 0; sz < contacts.size(); ++sz)
    {
      MailingList ml0 = generate(sz);              ❶
      MailingList ml1 = ml0; // copy constructor   ❷
```




```
      shouldBeEqual(ml0, ml1);                          ❸
    }
  {
    // Minimal check for deep copy - changing one should not affect the ot
    MailingList ml0 = generate(2);
    MailingList ml1 = ml0; // copy constructor
    ml1.addContact(jones);                              ❹
    BOOST_CHECK_EQUAL (2, ml0.size());
    BOOST_CHECK_NE (ml0, ml1);
  }
}
```

❶ We use the generate-and-check pattern.

❷ Here we invoke the function under test.




❸ The check function - we'll look at this in a moment

❹ The second half of this is more subtle. I expect that copies are distinct. Once a copy is made, updating one object should not change the other.

A failure suggests that we have done an improper shallow copy.

.....

**The Check Function for Copying**

```
// The fixture for testing class MailingList.
class MailingListTests  {
 public:
  Contact jones;
```




```
vector<Contact> contacts;

MailingListTests() {
  ⋮

void shouldBeEqual (const MailingList& ml0, const MailingList& ml1) cons
{
  BOOST_CHECK_EQUAL (ml1.size(), ml0.size());  ❶
  for (int i = 0; i < ml0.size(); ++i)         ❷
    {
  BOOST_CHECK_EQUAL(ml1.contains(contacts[i].getName()), ml0.contains(co
  if (ml1.contains(contacts[i].getName()))
    BOOST_CHECK_EQUAL(ml1.getContact(contacts[i].getName()), ml0.getCont
    }
```

```
  BOOST_CHECK_EQUAL (ml0, ml1);     ❸
  BOOST_CHECK (!(ml0 < ml1));
  BOOST_CHECK (!(ml1 < ml0));

  ostringstream out0;               ❸
  out0 << ml0;
  ostringstream out1;
  out1 << ml1;

  BOOST_CHECK_EQUAL (out0.str(), out1.str());

}
```




```
};
```

❶ Sizes should be equal

❷ Boths lists should agree as to what they contain.

❸ Relational ops - should be equal and not less/greater

❹ Whatever they print, it should match

.....

**Testing the Assignment Operator**

```
BOOST_FIXTURE_TEST_CASE ( assignment,  MailingListTests) {
```




```
for (unsigned sz = 0; sz < contacts.size(); ++sz)
  {
    MailingList ml0 = generate(sz);
    MailingList ml1;
    MailingList ml2 = (ml1 = ml0); // assignment    ❶
    shouldBeEqual(ml0, ml1);                        ❷
    shouldBeEqual(ml0, ml2);  // assignment returns a value
  }
{
  // Minimal check for deep copy - changing one should not affect the ot
  MailingList ml0 = generate(2);
  MailingList ml1;
  ml1 = ml0; // copy constructor
  ml1.addContact(jones);
```




```
    BOOST_CHECK_EQUAL (2, ml0.size());
    BOOST_CHECK_NE (ml0, ml1);
  }
}
```

- Very similar to testing the copy constructor

❶ But remember that assignment operators no only change the value on the left, but also return a copy of the assigned value.

  – We need to check both.

❷ Fortunately, we can re-use the check function developed for the copy constructor.



### Testing addContact

```
BOOST_FIXTURE_TEST_CASE (addContact, MailingListTests) {
  for (unsigned sel = 0; sel < contacts.size(); ++sel)
    {
      Contact& toAdd = contacts[sel];
      const Name& nameToAdd = toAdd.getName();
      for (unsigned sz = 0; sz < contacts.size(); ++sz)
        {
          MailingList ml0 = generate(sz);
          MailingList ml (ml0);
          bool alreadyContained = ml.contains(nameToAdd);
          ml.addContact (toAdd);
          BOOST_CHECK (ml.contains(nameToAdd));
```




```
        BOOST_CHECK_EQUAL (toAdd, ml.getContact(nameToAdd));
        if (alreadyContained)
          {
            BOOST_CHECK_EQUAL (ml.size(), (int)sz);
            BOOST_CHECK_EQUAL (ml0, ml);
            BOOST_CHECK (!(ml0 < ml));
            BOOST_CHECK (!(ml < ml0));
          }
        else
          {
            BOOST_CHECK_EQUAL (ml.size(), (int)sz+1);
            BOOST_CHECK_NE (ml0, ml);
            BOOST_CHECK ((ml0 < ml) || (ml < ml0));    // one must be tr
            BOOST_CHECK (!((ml0 < ml) && (ml < ml0))); // ...but not bot
```




```
            }
          ostringstream out;
          out << ml;
          BOOST_CHECK_NE (out.str().find(nameToAdd), string::npos);
        }
    }
}
```

- Similar to version developed earlier, but now we go systematically through all the accessors
  - Needed to add checks on relational operators and output operator




**And so on...**

We continue in this manner until done.

```
#define BOOST_TEST_MODULE MailingList test

#include "mailinglist.h"

#include <string>
#include <sstream>
#include <vector>

#include <boost/test/included/unit_test.hpp>
//#define  BOOST_TEST_DYN_LINK 1
//#include <boost/test/unit_test.hpp>
```



```
using namespace std;

// The fixture for testing class MailingList.
class MailingListTests  {
 public:
  Contact jones;
  vector<Contact> contacts;

  MailingListTests() {
    jones = Contact("Jones", "21 Penn. Ave.");
    contacts.clear();
    contacts.push_back (Contact ("Muffin Man", "Drury Ln."));
    contacts.push_back (Contact ("Holmes", "221B Baker St."));
```




```
  contacts.push_back(jones);
  contacts.push_back (Contact ("Wolfe", "454 W. 35th St."));
}

~MailingListTests() {
}


MailingList generate(int n) const
{
  MailingList m;
  for (int i = 0; i < n; ++i)
    m.addContact(contacts[i]);
  return m;
```




```
}

void shouldBeEqual (const MailingList& ml0, const MailingList& ml1) cons
{
  BOOST_CHECK_EQUAL (ml1.size(), ml0.size());
  for (int i = 0; i < ml0.size(); ++i)
    {
      BOOST_CHECK_EQUAL(ml1.contains(contacts[i].getName()), ml0.contain
      if (ml1.contains(contacts[i].getName()))
        BOOST_CHECK_EQUAL(ml1.getContact(contacts[i].getName()), ml0.get
    }

  BOOST_CHECK_EQUAL (ml0, ml1);
  BOOST_CHECK (!(ml0 < ml1));
```




```
    BOOST_CHECK (!(ml1 < ml0));

    ostringstream out0;
    out0 << ml0;
    ostringstream out1;
    out1 << ml1;

    BOOST_CHECK_EQUAL (out0.str(), out1.str());

  }

};

BOOST_AUTO_TEST_CASE ( constructor ) {
```




```
  MailingList ml;
  BOOST_CHECK_EQUAL (0, ml.size());
  BOOST_CHECK (!ml.contains("Jones"));
  BOOST_CHECK_EQUAL (ml, MailingList());
  BOOST_CHECK (!(ml < MailingList()));
}


BOOST_FIXTURE_TEST_CASE ( copyConstructor,  MailingListTests) {
  for (unsigned sz = 0; sz < contacts.size(); ++sz)
    {
      MailingList ml0 = generate(sz);
      MailingList ml1 = ml0; // copy constructor
      shouldBeEqual(ml0, ml1);
```




```
    }
  {
    // Minimal check for deep copy - changing one should not affect the ot
    MailingList ml0 = generate(2);
    MailingList ml1 = ml0; // copy constructor
    ml1.addContact(jones);
    BOOST_CHECK_EQUAL (2, ml0.size());
    BOOST_CHECK_NE (ml0, ml1);
  }
}

BOOST_FIXTURE_TEST_CASE ( assignment,  MailingListTests) {
  for (unsigned sz = 0; sz < contacts.size(); ++sz)
    {
```




```
    MailingList ml0 = generate(sz);
    MailingList ml1;
    MailingList ml2 = (ml1 = ml0); // assignment
    shouldBeEqual(ml0, ml1);
    shouldBeEqual(ml0, ml2);  // assignment returns a value
  }
{
  // Minimal check for deep copy - changing one should not affect the ot
  MailingList ml0 = generate(2);
  MailingList ml1;
  ml1 = ml0; // copy constructor
  ml1.addContact(jones);
  BOOST_CHECK_EQUAL (2, ml0.size());
  BOOST_CHECK_NE (ml0, ml1);
```




```
  }
}


BOOST_FIXTURE_TEST_CASE (addContact, MailingListTests) {
  for (unsigned sel = 0; sel < contacts.size(); ++sel)
    {
      Contact& toAdd = contacts[sel];
      const Name& nameToAdd = toAdd.getName();
      for (unsigned sz = 0; sz < contacts.size(); ++sz)
        {
          MailingList ml0 = generate(sz);
          MailingList ml (ml0);
          bool alreadyContained = ml.contains(nameToAdd);
```




```
        ml.addContact (toAdd);
        BOOST_CHECK (ml.contains(nameToAdd));
        BOOST_CHECK_EQUAL (toAdd, ml.getContact(nameToAdd));
        if (alreadyContained)
          {
            BOOST_CHECK_EQUAL (ml.size(), (int)sz);
            BOOST_CHECK_EQUAL (ml0, ml);
            BOOST_CHECK (!(ml0 < ml));
            BOOST_CHECK (!(ml < ml0));
          }
        else
          {
            BOOST_CHECK_EQUAL (ml.size(), (int)sz+1);
            BOOST_CHECK_NE (ml0, ml);
```




```
            BOOST_CHECK ((ml0 < ml) || (ml < ml0));    // one must be tr
            BOOST_CHECK (!((ml0 < ml) && (ml < ml0))); // ...but not bot
          }
        ostringstream out;
        out << ml;
        BOOST_CHECK_NE (out.str().find(nameToAdd), string::npos);
      }
  }
}

BOOST_FIXTURE_TEST_CASE (removeContact, MailingListTests) {
  for (unsigned sel = 0; sel < contacts.size(); ++sel)
    {
      Contact& toAdd = contacts[sel];
```




```
      const Name& nameToAdd = toAdd.getName();
      for (unsigned sz = 0; sz < contacts.size(); ++sz)
        {
          MailingList ml0 = generate(sz);
          MailingList ml (ml0);
          bool alreadyContained = ml.contains(nameToAdd);
          ml.removeContact (toAdd);
          BOOST_CHECK (!ml.contains(nameToAdd));
          if (!alreadyContained)
            {
              BOOST_CHECK_EQUAL (ml.size(), (int)sz);
              BOOST_CHECK_EQUAL (ml0, ml);
              BOOST_CHECK (!(ml0 < ml));
              BOOST_CHECK (!(ml < ml0));
```




```
        }
      else
        {
          BOOST_CHECK_EQUAL (ml.size(), (int)sz-1);
          BOOST_CHECK_NE (ml0, ml);
          BOOST_CHECK ((ml0 < ml) || (ml < ml0));    // one must be tr
          BOOST_CHECK (!((ml0 < ml) && (ml < ml0))); // ...but not bot
        }
      ostringstream out;
      out << ml;
      string outs = out.str();
      BOOST_CHECK_EQUAL (outs.find(nameToAdd), string::npos);
    }
}
```




```
}

BOOST_FIXTURE_TEST_CASE (removeContactByName, MailingListTests) {
  for (unsigned sel = 0; sel < contacts.size(); ++sel)
    {
      Contact& toAdd = contacts[sel];
      const Name& nameToAdd = toAdd.getName();
      for (unsigned sz = 0; sz < contacts.size(); ++sz)
        {
          MailingList ml0 = generate(sz);
          MailingList ml (ml0);
          bool alreadyContained = ml.contains(nameToAdd);
          ml.removeContact (nameToAdd);
          BOOST_CHECK (!ml.contains(nameToAdd));
```




```
        if (!alreadyContained)
          {
            BOOST_CHECK_EQUAL (ml.size(), (int)sz);
            BOOST_CHECK_EQUAL (ml0, ml);
            BOOST_CHECK (!(ml0 < ml));
            BOOST_CHECK (!(ml < ml0));
          }
        else
          {
            BOOST_CHECK_EQUAL (ml.size(), (int)sz-1);
            BOOST_CHECK_NE (ml0, ml);
            BOOST_CHECK ((ml0 < ml) || (ml < ml0));    // one must be tr
            BOOST_CHECK (!((ml0 < ml) && (ml < ml0))); // ...but not bot
          }
```




```
          ostringstream out;
          out << ml;
          BOOST_CHECK_EQUAL (out.str().find(nameToAdd), string::npos);
        }
    }
}

BOOST_FIXTURE_TEST_CASE ( merging,  MailingListTests) {

  MailingList ml0;
  ml0.addContact(contacts[0]);
  ml0.addContact(contacts[1]);
  ml0.addContact(contacts[2]);
```




```
  MailingList ml1;
  ml1.addContact(contacts[2]);
  ml1.addContact(contacts[3]);

  ml1.merge(ml0);
  shouldBeEqual(ml1, generate(4));
}
```

- Yes, BTW, I did discover quite a few bugs in my MailingList code while putting this together.




## 2.2 Testing for Pointer/Memory Faults

In our example, we did not write explicit tests for the destructor. Partly that's because the destructor is already being invoked a lot - every time we exit a function or a { } statement block that declares a local variable. So we have some good reason to hope that the destructor has been well exercised already.

Also, most of the effects of a destructor are hard to see directly. How do you check to see if memory has been successfully deleted?

**Testing for Pointer/Memory Faults**

Most destructors simply delete pointers, and pointer issues are particularly difficult to test and debug.

- One way that pointer/memory use can be tested is to employ specialized tools that

replace the system functions for allocating and deallocating memory (`alloc` and `free`) with special versions that

- keep track of all blocks of memory that have been allocated,
- whether those blocks have been freed,
- whether any block has been freed more than once, etc.

. . . . . .

**Tools for Testing Pointer/Memory Faults**

- Purify is a well known commercial package for this purpose, but is not cheap.
- I have had good results with a free tool called LeakTracer.



. . . . . .

# 3 Be Independent: Mock Objects

**The Lowly Stub**

- Most of what *Unit does is to provide a standard, convenient framework for writing *test drivers*.
- But unit testing often requires *stubs* as well
  - supplying values/operations to the module under test
  - processing outputs from the module under test
- We can avoid stubs by doing bottom-up integration




- reduces independence of test suites
- restricts development options

. . . . . .

**Where ADTs and Stubs Meet**

In some ways, ADTs and OOP make it easier to do stubs.

A *mock object* is an object whose interface emulates a "real" class for testing purposes.

Mock objects may...

- be simpler than the real thing
- stand-in for classes not yet implemented




- decouple test suites
- offer mechanisms for data collection, thus improving testability

. . . . . .

**Example: Mailing List**

In our MailingList tests, we relied on a *Contact* class.
Here is the real interface for *Contact*:

```
#ifndef CONTACT_H
#define CONTACT_H

#include <iostream>
#include <string>
```




```
#include "name.h"
#include "address.h"

class Contact {
  Name theName;
  Address theAddress;

public:
  Contact (Name nm, Address addr)
    : theName(nm), theAddress(addr)
  {}
```




```
Name getName() const   {return theName;}
void setName (Name nm) {theName= nm;}

Address getAddress() const     {return theAddress;}
void setAddress (Address addr) {theAddress = addr;}

bool operator== (const Contact& right) const
{
  return theName == right.theName
    && theAddress == right.theAddress;
}

bool operator< (const Contact& right) const
{
```




```
    return (theName < right.theName)
      || (theName == right.theName
      && theAddress < right.theAddress);
  }
};

inline
std::ostream& operator<< (std::ostream& out, const Contact& c)
{
  out << c.getName() << " @ " << c.getAddress();
  return out;
}
```




```
#endif

#ifndef NAME_H
#define NAME_H

#include <iostream>
#include <string>

struct Name {
  std::string last;
  std::string first;
  std::string middle;
```




```
  Name (std::string lastName,
        std::string firstName,
        std::string middleName)
    : last(lastName), first(firstName),
      middle(middleName)
    {}

  bool operator== (const Name& right) const;
  bool operator< (const Name& right) const;

};

std::ostream& operator<< (std::ostream& out, const Name& addr);
```




```
#endif


#ifndef ADDRESS_H
#define ADDRESS_H

#include <iostream>
#include <string>

struct Address {
  std::string street1;
  std::string street2;
  std::string city;
```




```
std::string state;
std::string zipcode;

Address (std::string str1,
     std::string str2,
     std::string cty,
     std::string stte,
     std::string zip)
  : street1(str1), street2(str2),
    city(cty), state(stte), zipcode(zip)
  {}

bool operator== (const Address& right) const;
bool operator< (const Address& right) const;
```




```
};

std::ostream& operator<< (std::ostream& out, const Address& addr);


#endif
```

.....

**What Do We Need for the MailingList Test?**

But in our tests, the only things we needed to do with contacts was

- create them with known names




- copy/assign them
- getName()
- compare them, ordering by name

. . . . . .

**A Mock Contact**

So we made do with a simpler interface that

- made it easier to create and check contacts during testing
- but would still compile with the existing *MailingList* code




```
#ifndef CONTACT_H
#define CONTACT_H

#include <iostream>
#include <string>

typedef std::string Name;
typedef std::string Address;

class Contact {
  Name theName;
  Address theAddress;

public:
```




```
Contact() {}

Contact (Name nm, Address addr)
  : theName(nm), theAddress(addr)
{}


Name getName() const   {return theName;}
void setName (Name nm) {theName= nm;}

Address getAddress() const     {return theAddress;}
void setAddress (Address addr) {theAddress = addr;}

bool operator== (const Contact& right) const
```




```
  {
    return theName == right.theName
      && theAddress == right.theAddress;
  }

  bool operator< (const Contact& right) const
  {
    return (theName < right.theName)
      || (theName == right.theName
          && theAddress < right.theAddress);
  }
};

inline
```




```
std::ostream& operator<< (std::ostream& out, const Contact& c)
{
  out << c.getName() << " @ " << c.getAddress();
  return out;
}


#endif
```

Replacing the *Name* and *Address* classes by simple strings.

. . . . . .

**OOP and Mocks**

Object-oriented languages make it easier to create some mock objects




- Declare the mock as a subclass of the real interface
- Override the functions as desired

Example: for an application that drew graphics using the Java *java.awt.Graphics* API, I created the following as a mock class:

```
package Pictures;

import java.awt.Color;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.Rectangle;
```




```
import java.awt.Shape;
import java.awt.image.ImageObserver;
import java.text.AttributedCharacterIterator;

/**
 * Special version of Graphics for testing that simply writes all graphics
 * a string.
 *
 * @author zeil
 *
 */
public class GraphicsStub extends Graphics {

    private StringBuffer log;
```




```
public GraphicsStub() {
    log = new StringBuffer();
}

public void clearRect(int arg0, int arg1, int arg2, int arg3) {
    entering("clearRect",
            "" + arg0 + ":" + arg1 + ":" + arg2 + ":" + arg3);
}

public void clipRect(int arg0, int arg1, int arg2, int arg3) {
    entering("clipRect",
            "" + arg0 + ":" + arg1 + ":" + arg2 + ":" + arg3);
}
```




```
    public void copyArea(int arg0, int arg1, int arg2, int arg3, int arg4,
            int arg5) {
        entering("copyArea",
                "" + arg0 + ":" + arg1 + ":" + arg2 + ":" + arg3 + ":" + a
    }

    public Graphics create() {
        entering("create", "");
        return new GraphicsStub();
    }

    public void dispose() {
        entering("dispose", "");
```




```
    }

    public void drawArc(int arg0, int arg1, int arg2, int arg3, int arg4,
            int arg5) {
        entering("drawArc",
                "" + arg0 + ":" + arg1 + ":" + arg2 + ":" + arg3 + ":" + a
    }

    public boolean drawImage(Image arg0, int arg1, int arg2, ImageObserver
        entering("drawImage",
                "" + arg0 + ":" + arg1 + ":" + arg2 + ":" + arg3);
        return true;
    }
```




```
    public boolean drawImage(Image arg0, int arg1, int arg2, Color arg3,
            ImageObserver arg4) {
        entering("drawImage",
                "" + arg0 + ":" + arg1 + ":" + arg2 + ":" + arg3);
        return true;
    }

    public boolean drawImage(Image arg0, int arg1, int arg2, int arg3,
            int arg4, ImageObserver arg5) {
        entering("drawImage",
                "" + arg0 + ":" + arg1 + ":" + arg2 + ":" + arg3 + ":" + a
        return true;
    }
```




```
    public boolean drawImage(Image arg0, int arg1, int arg2, int arg3,
            int arg4, Color arg5, ImageObserver arg6) {
        entering("drawImage",
                "" + arg0 + ":" + arg1 + ":" + arg2 + ":" + arg3 + ":" + a
        return true;
    }

    public boolean drawImage(Image arg0, int arg1, int arg2, int arg3,
            int arg4, int arg5, int arg6, int arg7, int arg8, ImageObserve
        entering("drawImage",
                "" + arg0 + ":" + arg1 + ":" + arg2 + ":" + arg3 + ":" + a
                + arg6 + ":" + arg7 + ":" + arg8);
        return true;
    }
```




```
public boolean drawImage(Image arg0, int arg1, int arg2, int arg3,
        int arg4, int arg5, int arg6, int arg7, int arg8, Color arg9,
        ImageObserver arg10) {
    entering("drawImage",
            "" + arg0 + ":" + arg1 + ":" + arg2 + ":" + arg3 + ":" + a
            + arg6 + ":" + arg7 + ":" + arg8 + ":" + arg9);
    return true;
}

public void drawLine(int arg0, int arg1, int arg2, int arg3) {
    entering("drawLine",
            "" + arg0 + ":" + arg1 + ":" + arg2 + ":" + arg3);
}
```



```
public void drawOval(int arg0, int arg1, int arg2, int arg3) {
    entering("drawOval",
            "" + arg0 + ":" + arg1 + ":" + arg2 + ":" + arg3);
}

public void drawPolygon(int[] arg0, int[] arg1, int arg2) {
    entering("drawPolygon",
            "" + arg0 + ":" + arg1 + ":" + arg2);
}

public void drawPolyline(int[] arg0, int[] arg1, int arg2) {
    entering("drawPolyline",
            "" + arg0 + ":" + arg1 + ":" + arg2);
```




```
    }

    public void drawRoundRect(int arg0, int arg1, int arg2, int arg3, int
            int arg5) {
        entering("drawRoundRect",
                "" + arg0 + ":" + arg1 + ":" + arg2 + ":" + arg3 + ":" + a
    }

    public void drawString(String arg0, int arg1, int arg2) {
        entering("drawString",
                "" + arg0 + ":" + arg1 + ":" + arg2);
    }

    public void drawString(AttributedCharacterIterator arg0, int arg1, int
```




```
        entering("drawString",
                "" + arg0 + ":" + arg1 + ":" + arg2);
    }

    public void fillArc(int arg0, int arg1, int arg2, int arg3, int arg4,
            int arg5) {
        entering("fillArc",
                "" + arg0 + ":" + arg1 + ":" + arg2 + ":" + arg3 + ":" + a
    }

    public void fillOval(int arg0, int arg1, int arg2, int arg3) {
        entering("fillOval",
                "" + arg0 + ":" + arg1 + ":" + arg2 + ":" + arg3);
    }
```




```
    public void fillPolygon(int[] arg0, int[] arg1, int arg2) {
        entering("fillPolygon",
                "" + arg0 + ":" + arg1 + ":" + arg2);
    }

    public void fillRect(int arg0, int arg1, int arg2, int arg3) {
        entering("fillRect",
                "" + arg0 + ":" + arg1 + ":" + arg2 + ":" + arg3);
    }

    public void fillRoundRect(int arg0, int arg1, int arg2, int arg3, int
            int arg5) {
        entering("fillRoundRect",
```




```
                "" + arg0 + ":" + arg1 + ":" + arg2 + ":" + arg3 + ":" + a
    }

    private Shape clip = null;
    public Shape getClip() {
        return clip;
    }

    public Rectangle getClipBounds() {
        return null;
    }

    private Color color;
    public Color getColor() {
```




```
        return color;
    }

    public Font getFont() {
        return null;
    }

    public FontMetrics getFontMetrics(Font arg0) {
        return null;
    }

    public void setClip(Shape arg0) {
        entering("setClip",
                "" + arg0);
```




```
        clip = arg0;
    }

    public void setClip(int arg0, int arg1, int arg2, int arg3) {
        entering("setClip",
                "" + arg0 + ":" + arg1 + ":" + arg2 + ":" + arg3);
    }

    public void setColor(Color arg0) {
        entering("setColor",
                "" + arg0);
        color = arg0;
    }
```




```
    public void setFont(Font arg0) {
        entering("setFont",
                "" + arg0);
    }

    public void setPaintMode() {
        entering("setPaintMode", "");
    }

    public void setXORMode(Color arg0) {
        entering("setXORMode",
                "" + arg0);
    }
```




```
    public void translate(int arg0, int arg1) {
        entering("translate",
                "" + arg0 + ":" + arg1);
    }

    public String toString()
    {
        return log.toString();
    }

    public void clear()
    {
        log = new StringBuffer();
    }
```




```
    private void entering(String functionName, String args)
    {
        log.append(functionName);
        log.append(": ");
        log.append(args);
        log.append("n");
    }

}
```

It basically writes each successive graphics call into a string, that can later be examined by unit test cases.

.....




**OO Mock Example**

Here is a test using that mock:

```
package PictureTests;
import java.awt.Color;
import java.awt.Point;
import java.lang.reflect.Method;
import java.util.Scanner;

import junit.framework.*;
import junit.textui.TestRunner;
import Pictures.GraphicsStub;
import Pictures.LineSegment;
import Pictures.Shape;
```




```
/**
 * Test of the LineSegment class
 */
public class LineSegmentTest extends TestCase {
    ⋮
    public void testPlot()
    {
        Point p1 = new Point(22, 341);
        Point p2 = new Point(104, 106);
        LineSegment ls = new LineSegment(p1, p2, Color.blue, Color.red);

        GraphicsStub g = new GraphicsStub();
```



```
        GraphicsStub gblue = new GraphicsStub();
        gblue.setColor(Color.blue);
        GraphicsStub gline1 = new GraphicsStub();
        gline1.drawLine(22, 341, 104, 106);
        GraphicsStub gline2 = new GraphicsStub();
        gline2.drawLine(104, 106, 22, 341);

        ls.plot(g);
        assertTrue (g.toString().contains(gblue.toString()));
        assertTrue (g.toString().contains(gline1.toString())
                    || g.toString().contains(gline2.toString()));
    }
    ⋮
```




- The (main) mock graphics object is created here.
- The function being tested in called here.
- Then, assertions like this one can test the info recrded in the mock object.

. . . . . .

**Mock Frameworks**

Increasingly, unit test frameworks are including special support for mock object creation.

- For example, the Google Mock Framework provides support for creating mock objects that automatically keep a log of calls made to them.




- And adds special functions and test assertions to both test that the expected calls were made and controlling what results will be returned from those calls.

  E.g., the following assertion says that a function `g.foo(x)` will be called 5 times, with $x$ being 1 and 2 one time each.

```
EXPECT_CALL(turtle, foo(_))
    .TIMES(3)
    .WillRepeatedly(Return(150));
EXPECT_CALL(turtle, foo(1))
    .WillOnce(Return(100));
EXPECT_CALL(turtle, foo(2))
    .WillOnce(Return(200));
```

  It also specifies what value should be returned on each call.

- If the function is called 6 times, or if one or more of the 5 expected calls are not




made before the test case ends, the `EXPECT_CALL` assertion fails, just like other test assertions.

. . . . . .

## 4 Be Pro-active: Write the Tests First

**Be Pro-active**

Ideally, we have made it easier to write self-checking unit tests than to write the actual code to be tested.

- The goal is to encourage a philosophy of writing the tests before wirting the code.

. . . . . .



**Debugging: How Can You Fix What You Can't See?**

The test-first philosophy is easiest to understand in a maintenance/debugging context.

- Before attempting to debug, write a test that reproduces the failure.
- How else will you know when you've fixed it?
- From a practical point of view, debugging generally involves running the buggy input, over and over, while you add debugging output or step through with a debugger.
  - You want that to be as easy as possible.
  - Doing this at the unit level, instead of in the context of the entire system, is often a dramatically better use of your time and effort.



. . . . . .

**Test-Writing as a Design Activity**

Every few years, software designers rediscover the principle of writing tests before implementing code.

Agile and TDD (Test-Driven Development) are just the latest in this long chain.

- Writing tests while "fresh" yields better tests than when they are done as an afterthought
- Thinking about boundary and special values tests helps clarify the software design
  - Reduces the number of eventual faults actually introduced
- Encourages designing for *testability*



– Making sure that the interface provides the “hooks” you need to do testing in the first place.

. . . . . .