Apps And Test Drivers, Reading

Once your class is written and fully tested, it's ready for use in real applications (or "apps"). What we did in this module so far is the writing part. But we never did any testing! All you have is the professor's word that this will all work when it's put together.

What we left out is all the small programming and testing steps that the professor did while you were distracted by the reading! In *this* reading we'll go through that process so you can see how the code actually got developed and tested to be sure it could reliably be presented to students in a data structures class.

Test Driver CPP

A "test driver" is a program that fully tests a class. But it's *not* written at the end of the class' development -- it's **written *during* development**. It checks that things are working correctly every small step of the way. Then if there is a problem of some kind, it will most likely be associated with the last bit of coding that was done, and you'll know where to look and apply debugging techniques.

Start with the world's smallest C++ program -- give it a name that identifies it as a test driver for the class, save, and compile. Here is **Array.TestDriver.cpp** and its command line compile command:

```
int main( )
{
}
```

```
prompt> c++ -std=c++11 Array.TestDriver.cpp -Wall
```

If that compiles, then you know your command line is located in the right "**working folder**" where your new CPP got saved. Note that the command line is prepared for **C++11 extensions**, and the -Wall is there so that the compiler will show **all W**arnings -- even little nit-picky ones that you can ignore.

If you are using an IDE, make sure that you are **working in "release" mode** and not in "debug" mode. The latter is more forgiving of logic errors (like overrunning arrays or wild pointers or references to no-longer-existing variables). The danger is developing in debug mode, finding all is okay, then delivering your product in release mode and finding it is *not* okay. Debugging that kind of problem is something you never what to have to do!

And just to *make sure* you are **editing the same file that you are compiling**, introduce a syntax error (like spelling "main" wrong), save and recompile -- if it catches the error, you're good.

Adding The class Definition

Now you can add the class' container above main, and any #includes that it may require -- none, in our case:

```
class Array
{
  int values[10];
  int dummy;

  public:
  Array( ); // constructor
  int capacity( ) const;
  int operator[ ](int) const;
  int& operator[ ](int);
};

int main( )
```

```
{
}
```

Save and compile and make any syntax corrections you need to make.

Adding And Testing Functions, One At A Time

As each function is added, *test it*. Devise a test that is thorough enough to **convince yourself that the function works** as it should under any circumstance (like deliberately overrunning an array in a range-safe data structure). Your test should include test input values decided by *you* -- not from cin or fin. It should include cout statements to output what you *expect* a result to be, and what is the *actual* result.

Don't delete any test code after you're sure it works -- leave it there, in case you make changes later that would require retesting. As such, your test program will grow, and its output will become longer and longer -- so long that it will be difficult for a human (you) to read it all, looking for mismatches between the actual and expected values. What we need is for the computer to **stop when it finds a mismatch**.

There are lots of ways to do that, including writing if-statements, but an easy way is to use "assertions". There is a C library named "cassert" that includes a simple void function named "assert" whose one parameter is expected to be true as a Boolean expression, or non-zero as a numeric expression. If it is, the program runs right through the statement. But if it's **false or zero, the program ends** right then and there, with some sort of crash message that's specific to your system.

Try this to see if your compiler is **configured properly to work with assertions**:

```
#include <cassert>

int main( )
{
  assert(false);
}
```

That *should* crash. If it does not, you'll need to find out how to activate assertions in your compiler.

Adding The Array::Array Main Constructor

The constructor initializes the data members -- remember that wanted all values initialized to zero. `int( )` is the default value for an `int` -- zero. Here's the function but there's no way to test it yet -- write it below int main:

```
class Array
{
  int values[10];
  int dummy;

  public:
  Array( ); // constructor
  int capacity( ) const;
  int operator[ ](int) const; // getter version
  int& operator[ ](int); // setter version
};

int main( )
{
  Array a;
}
```

```
Array::Array( )
{
  for (int i = 0; i < 10; i++)
    values[i] = int( );
}
```

Adding And Testing Array::capacity

Let's add and test **Array::capacity** -- note that the identifying comments and cout's are not included in this and the rest of the code samples in this modules and later ones, but you're still to put them in anything you submit for grading.

```
#include <iostream>
using namespace std;

#include <cassert>

class Array
{
  ...
  int capacity( ) const {return 10;}
  ...
};

int main( )
{
  Array a;

  // Array::capacity
  cout << "\nTesting Array::capacity\n";
  cout << "EXPECTED: 10\n";
  cout << "ACTUAL: " << a.capacity( ) << endl;
  assert(10 == a.capacity( ));
}

...
```

At this point we'll need assert and cout, hence the #includes. The function is written inline. An object for testing is created in main, and a code block is added to say the test's name, show expected and actual, and with an assertion to stop the program if things don't match up.

That's about it for this simple function. It's enough to convince that it works, and will work in all cases (because it's so simple).

Adding And Testing The Array::operator[ ] Setter

The setter is the preferred version by the compiler, which only ever uses the getter if it has to in case the Array object is const. For that reason, we're doing the setter first.

Add the function (below main), save and compile:

```
int& Array::operator[ ](int index)
```

```
{
  if (index < 0) return dummy;
  if (index >= 10) return dummy;
  return values[index];
}
```

Successful compiling tells you that the code has no syntax errors -- at least none that your compiler can detect! Now add a test block and include one for the constructor too (which we an do now):

```
  // Array::Array
  cout << "\nTesting Array::Array\n";
  for (int i = 0; i < a.capacity( ); i++)
    assert(a[i] == 0);

  // Array::operator[ ] setter
  cout << "\nTesting the Array::operator[ ] setter\n";
  a[6] = 12356;
  a[7] = 7654321;
  cout << "\nTesting Array::operator[ ] setter\n";
  cout << "EXPECTED: 12356 for a[6]\n";
  cout << "ACTUAL: " << a[6] << endl;
  assert(12356 == a[6]);
  cout << "EXPECTED: 7654321 for a[7]\n";
  cout << "ACTUAL: " << a[7] << endl;
  assert(7654321 == a[7]);
  a[-1000] = 123123;
  cout << "EXPECTED: 123123 for a[-1000]\n";
  cout << "ACTUAL: " << a[-1000] << endl;
  assert(12356 == a[6]);
  assert(7654321 == a[7]);
  assert(123123 == a[-6]); // any out-of-range uses dummy
  assert(123123 == a[10]); // checks upper end of range
  assert(123123 != a[9]); // checks upper end of range
  assert(123123 != a[0]); // checks lower end of range
```

Check out all the assertions -- they do additional checks with no couts. You should write enough of these to convince yourself that the setter function works as you expect it to. And yes, this uses the *SETTER*. We'll get to the as-yet-unwritten getter below.

Adding And Testing The Array::operator[ ] Getter

The getter returns a *copy* of an `int`, and not a reference, so just about any value you choose to return is fine -- a number (like zero below), a local variable, the parameter, or even `dummy`.

```
int Array::operator[ ](int index) const
{
  if (index < 0) return 0;
  if (index >= 10) return 0;
  return values[index];
}
```

To get the compiler to choose the getter for the testing, we need a const version of the Array object. Having already tested object copy, we can make

another, **const** copy:

```
// Array::operator[ ] getter
cout << "\nTesting the Array::operator[ ] getter\n";
const Array b = a;
for (int i = 0; i < 10; i++)
  assert(a[i] == b[i]);
```

With that, the public member functions are all fully tested. But there's one more test to do...

Const Object Test

The purpose of this test is confirm that the public member functions that we *intended* to be getters are in fact *coded* as getters -- that is, with a trailing **const**.

```
// const object test
cout << "\nConst object test\n";
const Array c; // if this compiles, Array::Array main constructor exists
assert(c.capacity( )); // if this compiles, Array::capacity is a getter
assert(c[0] == c[0]); // if this compiles, there is an Array::operator[ ] getter
assert(c[-1] == c[-1]); // tests the getter's range checking
```

Remember -- we're just checking the these lines *compile* without error. The compiler error you'll get if the trailing const was left off is pretty easy to understand -- this is your opportunity to make any corrections.

Complete List Of Tests -- An Initial Checklist

Here are the tests you'll do in any test driver:

1. Test all public functions to make sure they work as you expect them to work.
2. Const object test. (like `const Array c;`)

But there are three more tests to add to this list -- look for them in future modules as we learn and implement new coding features.

Variations

Our Array class is not particularly portable or reusable, because it's specifically for ten int values. The class' coding has to be modified for any other data type or capacity. We'll overcome those limitations in the next module, but if you're wondering why we did not write the class in a separate file and have a multi-file project for testing, it's because (1) it's not portable and will have to be rewritten for any variation, so it may as well be copied and pasted into new test drivers and apps, and (2) *after* this module we will have no need for having learned multi-cpp projects in *this* module, so why go to that trouble? Multi-cpp projects is an object-oriented programming topic -- look for that in DVC's Comsc-200 course.