Improving automated error diagnosis for OOP practice with C++

Pablo J. Novara

Universidad Nacional del Litoral, Santa Fe, Argentina
pnovara@unl.edu.ar,

Abstract. This paper describes how to exploit some particular rules about generic programming implementation with C++, to be able to verify at compile-time the existence of symbols such as classes, functions or methods, and the validity of expressions involving those symbols. This validation is done in such a way that the program will always compile and run, even if some expected class or function interface is missing, automatically skipping pieces of code when its compilation is not guaranteed to be successful. This techniques can be easily extended and applied to design and implement programming exercises that will precisely diagnose students errors when are executed, displaying custom run-time messages, and avoiding most compile time errors. This approach produces a self-contained C++11 code representing a programming assignment, that requires only a standard compliant compiler to be used by the student. No other tool or instrumentation is needed.

Keywords: programming teaching, object oriented programming, C++, generic programming, error diagnosis

1 Introduction

This paper will present some very simple tools to automate verification and qualification of little coding exercises for a basic C++ based object oriented programming course. All the diagnosis for an exercise answer will be done with more instrumented C++ code, so it will require nothing else but a standard conforming compiler. This is a very common practice for some particular sub-set of programming exercises, such as algorithmic ones. For instance, a teacher can ask the student to implement a function for sorting a vector, given an specific signature, and provide the student with a main function that calls that sorting function with predefined or random test cases and analyzes the function's outputs to determine whether the sorting algorithm is correct. This extra instrumented C++ code can be hidden to the student (through #include directives, obfuscation, or precompiled objects) to avoid confusion or cheating. The student only needs to care about its own syntax errors, and let the main program test for algorithmic ones. This paper will show how that kind of evaluation can be extended by some C++11 features to include situations where the student has to define the functions signatures, class interface, or data structures itself. The

presented mechanisms will not produce extra compiling errors, even when the interfaces the students defines (or not) make the testing code invalid. The student will still only see compiler messages related to his own syntax errors, not mixed with extra testing code ones. Furthermore, that extra code will be able to diagnose the student's design errors at run-time and with custom and very specific messages. Such a tool can be used to guide the student to the correct solution, or the score its performance on a test.

1.1 Application example

The following code shows an auto-contained programming exercise example. The first commented lines describe the task.

```
/// Assignment: implement a class Animal to represent
/// a pet. The class should have a constructor that
/// receives the pet's name and a method GetName()
/// to query the pet's name later.

// ...here goes student solution...

// include "test_solution.h"
```

The following listing shows `test_solution.h`. This code will evaluate the student's solution, checking for a suitable class interface, and testing an object's functionality:

```
#include "test_base.h"
class Animal;

// auxiliar types for testing class interface
// (test the validity of the third parameter's expression)
make_condition( animal_exists, Animal, sizeof(Animal));
make_condition( constructor_accepts_name,
                Animal, Animal("...") );
make_condition( can_report_name, Animal,
                string( ((Animal*)0)->GetName() ) );

// function that, assuming interface is correct,
// test its functionality
make_function(Animal, test_correct_name) {
  Animal ice_age_mammoth("Manny");
  string reported_name = ice_age_mammoth.GetName();
  if (reported_name!="Manny")
      exit("GetName class reports a wrong name!");
}

// main program, apply tests
int main() {
  if (animal_exists::is_false)
    exit("You have not defined Animal class");
```

```
  if (constructor_accepts_name::is_false)
    exit("The required constructor is not present");
  if (can_report_name::is_false)
    exit("GetName method is missing or wrong");
  test_correct_name<
    constructor_accepts_name::is_true &&
    can_report_name::is_true > ();
  cout<<"Congratulations, your code is correct!"<<endl;
}
```

Section 3 will discuss the content of `test_base.h` that makes this code possible.

1.2 Motivation

Every programming paradigm has a strong computer science theory behind, supporting its main high-level abstraction mechanisms and its actual low level implementation. But it is well known and accepted that learning programming requires a lot of coding practice, usually demanding much more time and effort than its supporting theory. Most programmers start coding with only a very restricted knowledge that theory, and acquire the understanding of the missing parts of the big picture later, through practice and by facing real world problems. So, Most programming courses include in some way or another a lot of coding practice. It is very common to teach programming basis based on ad-hoc restricted examples and simplified real-world situations.

A student needs to get some sort of feedback of a coding practice to make it really useful for its learning process. That feedback can be provided by different sources: a teacher inspecting the code, a compiler or a static analysis tool throwing error messages, the resulting program being executed and displaying right or wrong results, by an automated system testing or profiling its execution somehow, or a combination of them. Which one is the best one depends strongly on what is the intended purpose of that coding exercise. Compiler messages can usually only give feedback about syntax problem and some very common and generic implementation pitfalls. Automated tests and dynamic analysis requires a syntactically correct code and when that code is just a piece o a whole program (a very common case), it requires also some specific semantics for the symbols that the student defines on it (for instance, a particular signature for a function or an specific class hierarchy). The output of the resulting program based on students input requires both, and, as a basic testing method, it can only prove the existence of a bug, not its absence. Some sort of combined automated approach usually requires the implementation of some specific testing platform (in a way that resembles a coding contest).

Eventually, a teacher can replace all of them, but this can demand a big effort and sometime a too much of his time. There are many situation where it is not feasible. Sometimes, the the number of students per teacher makes it impossible. In other cases, a teacher can present the students a very big number of coding exercises, in order to provide the student a varied and bast set of

situations for home practicing. It is also very common to include some sort of continuous evaluation system. It means that the teacher will evaluate students progress very frequently (for instance, once a week) and so will have to read and grade many little pieces of code regularly. It is also a common problem the lack of time in a university quarterly course to developed all the expected course's topics, and so the teacher needs to dedicate as much time as possible to practice without degrading evaluation. Most of these scenarios are very common in many University's classrooms, and they can be even more important for non face-to-face teaching methodologies, such as e-learning or b-learning strategies.

All the presented reasons lead to the convenience to explore automated methods for coding-exercises' qualification, with clear and specific errors diagnosis based on the exercise's intended purpose, and demanding as little as possible of the student's environments and tools. The presented solution will let the teacher write custom messages for the potential students' errors (either very specific or general ones), and will require nothing else from student's environment but a standard C++11 complaint compiler, which is very becoming common nowadays (Microsoft Visual Studio 2013, gcc 4.8, llvm-clang 3.3, etc.), and should be already available in any C++ learning environment.

2 Generic programming with C++

This section describes some basic concepts about generic programming with C++ and will point some particular rules that will be exploded in the proposed solution in section 3. If you're familiar with C++ function overloading and templates concepts, partial and explicit specialization, and SFINAE rule, you can skip this whole section and continue reading section 3.

Generic programming is programming focusing on an algorithm and ignoring the particular data type where its going to be applied. The main goal is to raise the abstraction level and make algorithms and data structures more reusable. C++ has very powerful static type checking system, but generic programming is one of its main and most used features. Every variable or expression must have, either explicitly or implicitly, exactly defined its type at some point in the compiling process. This information allows the compiler to apply many optimizations that make good C++ code very efficient and performant compared to other languages. The tool provided by the language to allow generic programming is called template system (section 14 on C++ standard [1]):

```
template<typename T> void foo(T bar) { /*some code*/ }
...
do_something<float>(3.f);
do_something(3.f);
```

Previous code declares a generic function `foo`. To use it, we must provide an specific type that the compiler will use to replace `T`, produce what is called an specialization, and try to compile it. The example first calls it with `T=float`. In this particular case, such as most uses with functions where the generic type is among the function's arguments, it is not necessary to make `T=float` explicit.

The compiler can deduce it, as it does with the second call. The template argument can also be a constant of a given type instead a type name. Passing a constant value as a template argument instead of doing so as a regular function arguments is sometimes desirable because the former will be applied done at compile-time, while the latter at run-time.

```
template<int N> void foo() { int array[N]; /*some code*/ }
```

2.1 Partial and explicit specialization

For a given generic function or class, there's a way to implement special different versions for a particular type, or for a subset of all the possible types for the original one. The former is called "explicit specialization", the latter is called partial specialization.

```
// explicit specialization
template<int D> int remainder(int a) { return a%D; }
template<> void remainder<D>(int a) { }

// partial specialization
template<typename T> void foo(T bar) { /*some code*/ }
template<typename T> void foo(T *bar) { /*some code*/ }

// partial specialization
template<typename T, int N> class Foo { /* some code*/ };
template<typename T> class Foo<T,0> { /* some code*/ };
```

The first partial specialization reduce the set from all possible types to pointers. When called with a pointer will use the second version, when called with anything else will use the first one. The second kind of partial specialization in that example fixes on of the two templates arguments. Will be used only when the second argument is `N=0`. This second kind of specialization is only applicable to class templates, not to generic functions. Also note that the specialization is not required to match the original function/class declaration. In the example, the explicit specialization has a different return type.

2.2 Function overloading and name lookup

C++ allows the definition of multiple functions (or class' methods) all with the same identifier as name, but varying in its formal arguments (either type or amount):

```
void foo();
void foo(int x);
void foo(int, x, int y);
void foo(string s);
template <typename T> void foo(const void *ptr);
void foo(...);
```

All functions declared in that listing can be implemented in the same program. When calling a function, a C++ compiler will try to automatically deduce which ones can actually be used, based on types and amount of actual arguments, considering also implicit type conversions. All valid templates specialization are eligible too. There are several rules to decide what to do when more than one of them is suitable for a given call (see section 13.3 in C++ standard [1]). For the sake of this article, we will only recall that a function with the ellipsis operator (variable arguments, that C++ inherited from C, such as the last one in the example), are the functions with worst chance of being selected (lowest priority in the decision).

2.3 SFINAE

Finally, the most important language feature for this article's application is a rule that is informally known as SFINAE. This stands for "Substitution Failure Is Not An Error". This rule applies during overload resolution stage when compiling class or function templates: when substituting the deduced type for the template parameter fails, the specialization is discarded from the overload set instead of causing a compile error [2]. This is a very useful tool for template metaprogramming. Several overloaded template functions can be defined with different requirements on the types, and the compiler will just ignore the ones that requires something that a particular type cannot provide when its invoked. This feature is exploded in the C++11 standard "type_traits" library, and is the base for current partial implementation of the "concepts" idea (a language feature that Stroustrup proposed for standarization many years ago[3], but its syntax and implications are still being debated).

```
template<typename T, int N> struct Aux { typedef T type; };
template<typename T> int get_size(Aux<T, sizeof(&T::size)> &v)
    { return v.size(); }
template<typename T> int get_size(Aux<T, sizeof(&T::Size)> &v)
    { return v.Size(); }
template<typename T> int get_size(T &v) { return -1; }
```

The listing defines a struct `Aux` with two generic arguments: a type `T` and an integer `N`. Inside, a `typedef` retrieves `T`. The functions `get_size` try to specialize `Aux` with a given type, and the size of a class function pointer, which is obtained with `sizeof` applied to an expression that will only be valid if `T` has a method named `size` or `Size`. If the function is invoked with a type that does not has either `size()` nor `Size()`, both specializations will fail, and the third one will be used. The only little problem with this approach, is that the automatic type deduction won't work as desired, so the user must call the generic function explicitly including the template argument.

3 Verifying symbols existence and expressions validity

No we'll explode the technique shown on section 2.2 and overload resolution rules to get a constant bool representing the existence of a symbol or the validity of an expression.

3.1 Testing for an attribute/method name's existence

Consider the following class for testing if some class definition has a `size` method:

```
template<typename T> struct HasSizeMethod {
    template<int N> struct Aux{};
    template<typename U> static int Test(
                                Aux<sizeof(&U::size)>* ){}
    template<typename U> static char Test(...){}
    static const bool is_true =
                           sizeof(Test<T>(0))==sizeof(int);
    static const bool is_false =
                          sizeof(Test<T>(0))==sizeof(char);
};

int main() {
   if (HasSizeMethod<x>::is_true) { /*some code*/ }
}
```

The key for this code to work is having two overloads for a `Test` method. One will always work (the one with variable arguments), the other will only works when the requirement on the type is satisfied. To change what is tested just change the `sizeof(...)` expression. When both `Test`'s specializations are valid (that can be the case when methods are invoked with 0 (convertible to `int` or `NULL` pointer), the first one will always be selected (anything beats variable arguments on overload resolution stage). Notice also that both methods return different types. It does not mater the actual return value, but only the type. Both types must have different sizes, so applying `sizeof` to a call to `Test` and comparing this with `sizeof` one of the types produces a constant boolean value that is evaluated at compile time. Finally, everything is made static, so the client program can use that result without actually constructing an object. We should define a class like this one for every symbol or expression we want to test.

3.2 Generalizing the testing expression

Anything that is accepted by `sizeof` can be used with the technique just presented to build a class template that tests its validity. But `U::size` being valid means that something called `size` exists withing the class `U` and is visible. But it may not be useful, depending on if it is indeed a method, what arguments does it accepts, what is its return type, and more. We might change the `int` argument in template `Aux` for `typename`, and try to invoke it from `Test` with a very specific function pointer signature. But again, most of the time we want

to know if we can use the object for something, but that "something" can be implemented in different ways. In the example of the `size` method, the return type could be `int`, `long`, `unsigned int`, or some kind of int-like object, and still be usefull. The method could be `const` or not. And there are many other variations that won't affect the object's usability, but will prevent the method signature from matching what the testing class requires. So, its better to test for an expression that actually uses the method as expected.

For instance, we can test for `sizeof(int(((U*)0)->size()))`. `(U*)0` builds a `NULL` pointer of type `U`. We use a pointer to avoid making assumptions about `U`'s constructors. Then we try to invoke `size` method without arguments and finally check if whatever it returns is convertible to `int`. This kind expression can be inside a `sizeof`, as long as the return type is not allowed to be `void`. `sizeof` cannot be applied to `void`, so we need a sightly different approach. C++11 introduced the keyword `decltype`, that acts like an special compile time function similar to `sizeof`, that receives some arbitrary expression and returns the expression's type. It is equivalent to use "`float f=1.5f;`" or "`decltype(1.5f) f=1.5f;`". Furthermore, `sizeof` returns an `std::size_t` object, son it can be used inside `decltype`. That way, we can use `decltype` for all the same tricks as `sizeof`, and some extra cases too. The second good property of using `decltype` for everything is that the helper class (in the example, HasSizeMethod) will always use a `typename` as argument for its `Aux` struct (where `sizeof` version used to require an `int`), so all testing classes will have exactly the same structure, and vary only on the tested expression (in previous version, the exact signature test required a different `Aux` class).

Now we can test the existence of a method with "`decltype(&T::Method)`", the method's usage with expressions such as "`decltype(string(((U*)0)->GetName()))`", or a class existence with "`decltype(sizeof(ClassName))`". In order to avoid a compiler error, we can make a forward declaration of the class name before testing. If the class was already defined, the forward declaration does nothing, and `sizeof` return its size. If the class was not defined, the forward declaration is enough to avoid the compiler error while testing it, but not enough to make `sizeof` applicable (the type is still incomplete).

3.3 Conditional compilation

We can easily use the result of the presented tests as run-time expressions to show different messages. But the whole proposed idea include using that to decide if a function that required such tests to be passed in order to be compiled, should actually be compiled. To do so, we use a generic function whose template argument is a boolean value and does nothing, and a explicit specialization for the value `true` to insert there all the class client code. Since the function should have the `typename` as template arguments to actually avoid errors when compiler parses the explicit version, this template will actually have two arguments (the type and the `bool`) and we will want to explicitly specialize only one of them. It it's no possible with functions (it's actually a partial specialization), only with classes. So we build a class and use its constructor as a function:

```
template<typename T, bool valid> struct foo { };
template<typename T> void foo<true> struct foo {
    foo() { /*some code that use T*/ }
};
```

3.4 Simplifying tests definition and invocation

As was remarked before, we need to define a new test class for every property of every class we want to test. And we need to define a generic empty class and an a partial specialization for every client code that actually tests the class functionality when interfaces are correct. Here we present two preprocessor functions to make these definitions much more easier:

```
#define make_condition(name,type,expression) \
    template<typename T> struct AuxClass##nombre { \
        template<typename U> struct Aux{}; \
        template<typename clase> static int Test( \
            Aux<decltype( expression )>* ) {return int();} \
        template<typename U> static char Test(...) \
            {return char();} \
        static const bool is_true = \
            sizeof(Test<type>(0))==sizeof(int); \
        static const bool is_false = \
            sizeof(Test<type>(0))==sizeof(char); \
    }; \
    typedef AuxClass##name<type> name

#define make_function(type,name) \
        template<typename T, bool B>  \
            struct AuxClass##name{}; \
        template<typename T> struct \
            AuxClass##name<T,true> { AuxClass##name(); }; \
        template<bool B> using nombre =  \
            AuxClass##name<clase,B>; \
        template<typename T> \
            AuxClass##name<T,true>::AuxC##name()
```

First one declares an auxiliary generic class with the test and a `typedef` to call it without repeating the specific desired type. Second one declares the generic empty class that will act as a client function, the partial specialization and a `typedef`-like alias (again, to avoid the need of repeating the tested type name on invocation), and move constructor's implementation outside class definition to let the preprocessor function invocation resemble a function definition. Here we use a `using` directive instead of `typedef` because `typedef` cannot be generic(cannot be combined with `template` to keep one or more template argument still unspecified). This meaning for `using` is new in C++11. Finally, in both functions the typename generic's identifier matches the identifier of the user type that is going to be tested. This hides the template complexity when invoking

that preprocessor function, since a generic name such as `U` would have forced the user to express the expression to test (the one for `decltype`) using that generic name `T` instead of the actual class name that he wants to test. Placing these two preprocessor functions in "test_base.h" header file, among with a trivial custom `exit` function, the example code presented in introduction becomes perfectly valid and complete standard C++11 code.

4 Conclusions

The combination of techniques developed in section 3 sets the basis for expanding the universe of C++ programming problems that can be self-contained in source code. The solution replaces the generic and sometimes cryptic compiler errors that can emerge due to flawed designs or implementations for a required set of class and/or functions, with customizable run-time messages that a teacher can predefine. This can be exploded to generate guided exercises and assignments that can be automatically graded by just compiling and running them. The method only requires a valid C++11 compiler to be applied. That compiler is supposed to be already available for C++ student, usually within a whole IDE. So, the student won't need to install extra tools, nor to communicate with testing servers or to learn/use a new coding environment. Compiler errors will be reduced to errors contained inside student's code. The preprocessor functions presented allow the teacher to easily create new exercises and define custom tests for class hierarchies and functions' interfaces with a very few lines of code. This tool explodes many particular rules and tricks of the C++ language in order to supply the lack of useful alternative mechanisms present in other languages (such as reflexion). It is intended to apply on programming courses where C++ is the first language the student sees, maybe preceded only by a short pseudocode stage, a situation not very uncommon in Latin American universities [1]

References

1. Standard for Programming Language C++, Working draft, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3376.pdf
2. cppreferece.com: SFINAE, http://en.cppreference.com/w/cpp/language/sfinae Accessed: 08-08-2014.
3. Andrew Sutton , Bjarne Stroustrup, Design of Concept Libraries for C++. Proceedings of the 4th international conference on Software Language Engineering, p.97-118, July 03-04, 2011, Braga, Portugal
4. Vandevoorde, David, Nicolai M. Josuttis, C++ Templates: The Complete Guide. Addison-Wesley Professional. ISBN 0-201-73484-2. (2002).

[1] The author of this article is also the developer and maintainer of a Spanish pseudocode based learning tool (PSeInt). A survey was conducted among institutions submitting pseudocode profiles for this tool, and that survey revealed that C++ was the second most popular language for teaching after pseudo-code (with 18%), only surpassed by Java (27%), and followed very close by C(17%), and not so close by Python (9%), from a total of 131 answers at the moment of writing this article.