Annales Mathematicae et Informaticae
37 *(2010) pp. 211–223*
http://ami.ektf.hu

C++ exam methodology

Norbert Pataki, Zalán Szűgyi

Department of Programming Languages and Compilers
Eötvös Loránd University

Submitted 20 October 2009; Accepted 9 March 2010

Abstract

The C++ programming language supports multiparadigm programming. We can write programs in procedural, object-oriented, generic way at the same time.

However, it is difficult to figure out exercises for the terminal examinations since not easy to separate the algorithmic cogitation from the knowledge of the programming language. There are some basic elements that programmer students have to know: constructors, parameter passing, objects, inheritance, standard library, handling constants, copying objects, functions and member functions, etc. Exercises must be multiparadigm according to the C++ language. Using only one paradigm in C++ is not enough. This results in that we have to distinguish the different linguistic constructs on the basis of its complexity.

Many questions are arisen in connection with the exercises of terminal examinations. How can we gauge the procedural, the object-oriented, and the generic paradigms at the same time? How can we gauge students' C++ knowledge when we do not lay stress on the algorithmic cogitation? What kind of exercises may be interesting by the Standard Template Library? Which C++ constructs are reckoned to be more difficult and which ones considered to be easier? What are the most important ones? In this paper we give answers to the previous questions, we describe our methodology to assessment of students' C++ knowledge in a semi-automatic grading way. We also present exercise examples that worked out according to our methodology. We take stock of students' results in the paper.

Keywords: C++, exam, teaching, multiparadigm programming

MSC: 68N19

1. Introduction

In software technology a *paradigm* represents the directives in creating abstractions [21]. The paradigm is the principle by which a problem can be comprehended and decomposed into manageable components. A paradigm directs us in identifying the elements in which a problem will be decomposed. The paradigm sets up the rules and properties, but also offers tools for developing applications.

C++ is usually considered as an object-oriented programming language, but it is not completely true. C++ supports *multiparadigm programming* [24]. Structured programming features come from C legacies with better parameter-passing opportunities and features of overloading. Classes may be created in a sophisticated way, for example the C++ programming language distinguishes between three different variants of inheritance based on access control. Templates are also supported. Generic and generative programming have become available with C++'s template construct. The *C++ Standard Template Library* (STL) was the very first library based on generic programming and its usage is similar to the functional programming approach [2], [18].

C++ is considered as a language that hard to teach. C legacies must be known because of their hazard but Stroustrup argues for a use of C++ as a higher-level language that relies on abstraction to provide elegance without loss of efficiency compared to lower-level styles [23]. Many paradigms and approaches should be taught at the same time. By the way, C++'s standard library is roomy. Standard Template Library (STL) includes more than sixty algorithms and seven actual containers and three adaptors. STL is just a part of the standard library.

Multiparadigm software design and its implementation in the C++ programming language are deeply investigated by James Coplien [7]. One of his most important conclusions is that different kind of domain problems should be targeted using different programming paradigms. The domain analysis, especially identifying positive and negative variability helps to select the most appropriate paradigm.

We work with only standard C++, so we do not deal with multithreaded C++ programs, sockets, graphical user interfaces. This can ease the teaching process as well as the examination. For example, we do not have to work with graphical forms, inputs and outputs which could not be integrated to our framework easily.

However, gauging students' C++ knowledge is much more harder. Students' attainments must be examined from many aspects.

Many elementary constructs can be found in the C++ programming language. All students should know these features: functions, classes, methods, templates. Students must use these constructs in a sophisticated way. Constructs like parameter passing, constructors, constants, copying objects, inheritance are also very important. Contrarily, importance of algorithmic cogitation should be minimalized because we gauge the C++ knowledge.

Teaching the standard library is important in a C++ programming language course [25]. Therefore the students can use the STL when writing exams. Many exercises are unusable, like lists, maps, vectors, etc. without significant modification

in their specifications.

In Hungary, a five-point grade system is used. 1 is the failing grade and 5 is the best possible grade. C++'s constructs should be reflected in this grade system. Which constructs the students must know, and which ones are more difficult? Which constructs are the most weighty ones?

In this paper we describe our methodology to assessment of students' C++ knowledge. We present the structure of former exercises that able to grading students in a semi-automatic way and an archetypal exercise is detailed.

This paper is organized as follows. In section 2 we describe the general conditions and introduce the frame of exercises. In section 3 we detail a specific exercise. Other ideas are presented briefly in section 4. We give a brief overview about our experiences in section 5. We analyze the students' results in section 6. Finally, we conclude our issues in section 7.

2. Exams in a nutshell

In this section we describe the general circumstances in connection with the exams.

Students have to write exams in a computer lab. They may use their books, notes and the world wide web, but they have to work alone. The exams last about 3 and a half hours. An experienced C++ programmer is able to solve these problems within half an hour.

Five different grades are distinguished in the Hungarian education. The grades denote in numbers from 1 to 5, where 1 means unsatisfactory and 5 means first.

The exercise is typically the implementation of a class template, with many member functions. Students receive the client code that instantiates the template. The client describes the specification of the class template as a sequence of use cases. For the pass grade the students have to implement some base functionality of class template. For better marks they need to implement more and more functions. At the beginning all the code of functional tests are commented. When the students implement all the necessary functions for a given mark they can uncomment the corresponding part of client code to see whether their work were correct or not. On the other hand, these functional and semantical tests are not given proof of the correctness of implementation but usually a very good feedback to the students [15].

The program always prints to display the student's mark, if the program can be compiled. When students download the exercise program, it displays the unsatisfactory mark. In our case students must progress linearly.

The main goal of exercise is usually a template container similar to STL's containers. The representation of the class is not determined. Students can freely choose the representation. The effectiveness is not a primary goal here, however extremely poor design is rejected.

Students have to write a template class with proper template parameters, a trivial constructor. Inserting elements must be supported and a basic information should be obtained from an object and a constant object. Copying object via copy

constructor and assignment operator is usually also needed to the pass mark. We reckon that these constructs are essential ones.

Usually the class must be extended for a better grade. The usual constructs for a fair mark are more difficult methods, like erasing elements, etc..

For a good or an excellent grade `iterator` or `const_iterator` inner type is often required. Iterator objects must work together with the STL algorithms. Students should use the STL containers and iterators to overcome this exercise, because they do not know all necessary members for iterator types.

For a good mark operators that *overloaded on const* are fine. Sometimes usage of polymorphism appears here.

For excellent grade clearly many constructs can be gauged. Special template constructors for any iterator types or template copy constructors are ideal. Sometimes *generic algorithms* are required that are not in the standard, like `copy_if`. These template algorithms must be similar to STL's algorithm. Introduction of a new template parameter with default value is reasonable. Basic template metaprogramming features (like overloading on the returning values according to a template argument) is also proper. We assume that students can take advantage of the STL.

Our method can be applied when the marking conditions are more complex.

The following excerpt describes the general schema of our exercises:

```
#include "work.h"
#include <iostream>

// necessary classes and functions

int main()
{
  int yourMark = 1;
  /* 2
  here we use the basic methods of the class: some use cases
  ...
  if the implementation suits the use cases, variable
  yourMark is increased
  */

  /* 3
  here we use more methods: more uses cases
  ...
  after some basic functional test, value of yourMark is 3
  */

  /* 4
  More difficult methods tested at this block:
  ...
  inasmuch as implementation passes the test, variable
```

```
  yourMark is increased
  */

  /* 5
  Quite difficult methods required in this block:
  ...
  after successful test cases, value of yourMark is 5
  */

  std::cout << "Your mark is " << yourMark << std::endl;
  return 0;
}
```

Students must use this schema, they are only allowed to uncomment the different parts. However, the linearity is not necessary, but we apply it. The different parts could be independent and at the end of parts variable can be increased one by one.

Students present their solution at the end of the exams. One of the teachers analyzes someone's code and asks the student to make sure cheatless. The teacher gives the grade based on the program's output, but he can give different grade. So, the students do not achieve the program's output as a grade automatically. This is important because the tests are not all-inclusive ones and it could be eluded. The structure of the exam makes much more easier the teacher's work and he or she can focus on the details and it is also a good feedback to the students.

In this section we introduced the general frame of exercises. We categorized the different linguistic constructs to gauge. Hereinafter we paraphrase a specific example that describes an exercise of sorted list template.

3. A detailed example

In this example a sorted list container must be implemented which is template. It keeps its elements ordered. Its public interface is quite similar to STL's list container, but STL's list container is not ordered.

The test file includes a functor class that called Compare for a user-defined comparison:

```
#include "sl.h"
#include <deque>
#include <iostream>
#include <string>
#include <numeric>
#include <functional>
#include <vector>
```

```
struct Compare: std::binary_function<int, int, bool>
{
  bool operator()(int a, int b) const
  {
    return a > b;
  }
};
```

Students must working in the file called sl.h. They must know that templates do not compose compilation units.

The following part must work to pass the exam. If this part does not work, then the student fails.

```
  /* 2
  SortedList<int> li;
  SortedList<double> ls;
  ls.insert(5.6);
  ls.insert(3.2);
  li.insert(7);
  li.insert(2);
  li.insert(5);

  const SortedList<int> cli = li;
  if (3 == cli.size())
    yourMark = cli.front();
  */
```

Default constructor must be callable. Inserting elements is required, and it should be an actual template: insert must work proper according to the template argument. Creating copy via copy constructor must be supported. This not a problem, if the standard list container is used for representation, because the default copy constructor calls the members' copy constructors to create copies. Furthermore. two more methods must be implemented: the size method that returns how many elements are in the list, and the front method the returns the list's very first element. These two methods called on constant list, so these are const methods according to the features of C++'s constant correctness. The very first element is least element in the ordered list, therefore value of `yourMark` variable will be 2. This part should not be a real challenge for prepared students: it is a very basic linked list. Of course, some students present worser accomplishment because of jitter. We try to help these students with more help or comment.

```
  /* 3
  li.insert(8);
  li.remove(5);
  if (7 == cli.back())
    yourMark = cli.size();
```

```
*/
```

Two more methods needed for fair grade: a remove method that erases a given element from the list, and a back method that returns list's last element.

Overload on const is not good in this specific example because an overloaded function would violate the constraint of orderness.

```
/* 4
const int N = std::accumulate(cli.begin(), cli.end(), 0);
yourMark += (14 == N);
*/
```

An iterator type is required for good grade. We call STL's accumulate algorithm with SortedList's iterator. Accumulate adds together the elements in the container. Therefore iterator's proper work is needed, because students cannot modify the accumulate algorithm. If accumulate returns 14, `yourMark` variable is increased. Implementation of an iterator class is not easy because many operators must be implemented and many special members are needed. But when STL's list is used for the representation this implementation is unnecessary, because we can use list's iterator instead of a handcrafted one.

```
/* 5
std::deque<int> d;
d.push_back(2);
d.push_back(1);
d.push_back(3);

const SortedList<int, Compare> lc1(d.begin(), d.end());

std::vector<int> v;
v.push_back(3);
v.push_back(7);
const SortedList<int, Compare> lc2(v.begin(), v.end());

if (7 == lc2.front())
  yourMark = lc1.front() + lc2.size();
*/
```

Two special features needed for the best grade. Arbitrary ordering can be passed as template argument by functor class. All previous code must be compiled with this feature, therefore the new template parameter needs default parameter. With the introduction of this parameter the list's behaviour must remain the same. `std::less<T>` is a standard functor class template to describe the normal behaviour of list ordering. The operator() of this template functor class calls the operator$<$ of T. Implementation of `less` is quite easy, but can be found in the

STL. This functor class must be the default argument to the new template parameter. Another feature is the special template constructor for arbitrary iterator types. In the example we use this constructor with `vector<int>::iterator` and `deque<int>::iterator`. All standard containers offer this kind of constructor. Nevertheless, overloading is not allowed to overcome this situation.

The functional tests in the previous code fragments do not ensure the correctness of the implementation. However, most problems can be discovered by this method.

This example is not too difficult from the view of algorithmic cogitation, but it is more and more difficult from the view of C++ language. This example presents our conception aright.

4. Other examples

We create our exercises according to our methodology. The previous example presents our ideas. We expect an implementation of a template class with ever more difficult features. We keep track the student's grade in a variable. This variable depends on the correct implementation of the exercise.

Many ideas can be found in [18]. The usual exercise is based on STL's flaws. Containers for pointers are not supported by the standard library. Containers of pointers cause many problems (for example, copying is not trivial and avoiding memory leaks).

Another flaw is STL's multimap container does not define the relative order of elements at the same key. A multimap container that defines the relative order of element at the same key is a fine exercise.

STL does not include hashing containers. Hashtables, hashmaps are also ideal containers for exams.

Caching associative containers are similar to the standard associative containers. They are sorted, they can take advantage of sortedness, ensure iterators, but they have a special invariant, their size is limited. If the container would be oversized, it erases the oldest element from the container. Any kind of these containers is good for exams.

Graph types also cannot be found in the standard library. Graphs are worth considering, because they can be gauged in many different ways.

Union of akin containers (for instance set and multiset, or stack and queue) can be worked out. The behaviour of union's container based on a bool template argument.

In this section we present some more examples in a nutshell. These ideas were the basis of former exams.

5. Experiences in general

In this section we present our general experiences in connection with students, exercises and C++ itself.

Typically, every kind of grades is achieved. The grades are harmonized with the students' capability. The main approach (selection of the representing object) determines the obtainable grades considerably.

One of the major experiences that STL mightily makes the examination's solution easier. STL allows students concentrating on linguistic challenges. This is the very same experience when STL allows professional programmers concentrating on runtime complexity and different optimizations. The better grades are reached almost only with the STL. For the best mark we assume that students use the STL, and no student can solve the last part without the library.

Strictly speaking, some of the students do not use the standard library, and implement a handcrafted linked list class (or other node-based container). These approaches often fail on small pitfalls. Special analysis tools (like valgrind) are not necessary to avoid memory related bugs. Typically handcrafted containers are makeshift and should be avoided in this situation.

6. Quantitative Results

In this section we present the results of students. First, we give an overview about the results of given semester chronologically regardless of resits. We also present the results in graphical way (see Figure 1, Figure 2 and Figure 3). These charts present the number of students who achieved the given grade.

7–10 students failed on every examinations in the examined semester. In addition there were some students who applied for an exam, but did not come.

Twenty-five students gained rather good (excellent and good) grades and eight students failed on the first occasion from fourty-six students. Presumably we claimed typical constructs for these marks. Generally, the more talented students come on the very first occasions. However, when a new series of datastructues are introduced we work out a lighter exam (easier member methods with easier algorithms) to focus on the new features. We keep our exams available on the local network.

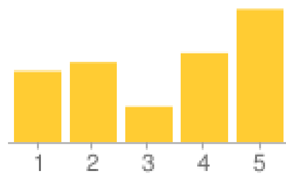

Figure 1: The results on the first time in a semester

The results of second occasion differ from the first one. Fifty-five students came to the exam, nine students failed. Twenty students reached the best grade. The differences became sharper, because the students could prepare for the examination

on the grounds of the previous exam and some of the students practiced with the previous exam, and some of them do not. These two exercises were similar, but the second one was more difficult. This approach results in this far cry.

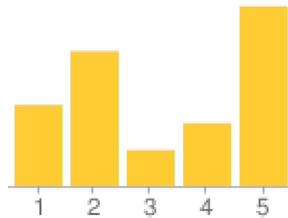

Figure 2: The results on the second time in a semester

Completely different result is yielded on the third time. Figure 3 looks like normal distribution, this result denotes correct exercise: only nine students were able to carry through the exam of fifty-five people, and ten of them cannot do the examination. Most of them – fourteen to be exactly – reached the better grade. The main reason of this incident is that we cannot continue of the previous series but we worked out a new exercise, that had a good difficulty level.

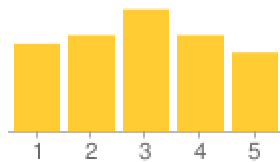

Figure 3: The results on the third time in a semester

Next, we consider results of more then thousand students from the last four years. We compare students' results to their results of *Ada programming language*. This couse is similar to ours, but students get the grade in a more classical way, teachers read through the student's code in the course of Ada programming language. Fails are not taken into account.

We divide the students into six groups. The first group is the students, who have rather good grades (excellent or good marks) from C++ as well as from Ada. This group includes 281 students. The second group is the students, who have rather bad grades (pass or fair marks). There are 421 people in this group. The third group includes the students who have rather good grades from C++ but have rather bad grades from C++. This group includes 192 students. The fourth group is just the opposite of the third one, this group contains the students, who have rather good grades from Ada but have rather bad grades from C++. 119 students are in this group. However, the third and fourth group contain students who have

fair mark from the either of the courses and better from the other one. This is not a significant difference. The students with significant difference can be found in the fifth and sixth group. Students who have excellent mark from C++ but have only pass mark from Ada are in the fifth group. This group contains 32 people. Students who have excellent mark from Ada but have pass grade from C++ are in the sixth group. 18 people belong to this group. The following chart presents these numbers in a graphical format.

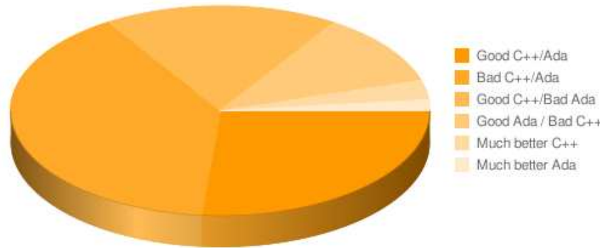

Figure 4: Connection between Ada and C++ grades

These numbers ensure our methodology is fair. About half of the students get similar grades from C++ and Ada but the exam methodology is quite different. Only few students have achieved completely different grades from the two courses. Special prolegomena (e.g. industrial experience) may causes these incidents.

In this section we argued for our methodology in a quantitative manner. We counted how many students achieved different marks in an entire semester. We compare our methodology to an other similar course's methodology. Our methodology has been confirmed by the computation.

7. Conclusion

The C++ programming language is difficult to teach and to learn. C++ supports multiparadigm programming. Functions, classes, generative constructs can be used in an orthogonal way.

However, contrive exams is much more harder process. In Hungary a five-grade system is used. We present our methodology based on this grading system. Our methodology supports multiparadigm programming. Our examples take advantage of STL's flaws and supports a semi-automatic grading system. This semi-automatic system means that our client code offers a mark that we check at the end of examination. The offered mark is based on test cases. The test cases are not all-inclusive but we give a feedback to the student as well to the teachers.

We presented our general framework to gauging students' knowledge and a specific example is detailed. We defined classification of C++ constructs based on

their difficulty and essentiality. We outlined students' results in a given semester. We compared the results of our excercises to the results of the course Ada programming language that applies a more classical method. Our charts confirm that our framework is fair.

References

[1] Assassa, G., Mathkour, H., Al-Ghafees, B., Automated Software Testing in Educational Environment: A Design of Testing Framework for Extreme Programming, *First National IT Symposium (NITS2006), Bridging the Digital Divide: Challenges and Solutions* (2006).

[2] Austern, M. H., Generic Programming and the STL, *Addison-Wesley* (1999).

[3] Cardelli, L., Wegner, P., On Understanding Types, Data Abstraction, and Polymorphism, *ACM Computing Surveys*, 17(4), (1985) 471–522.

[4] Cifuentes, C., Brannan, B., Teaching C/C++ to Computer Science Students with Pascal Programming Experience, *Proceedings of the 1st Australasian conference on Computer science education* (1996) 189–196.

[5] Colton, D., Fife, L., Thompson, A., A Web-based Automatic Program Grader, *Information Systems Education Journal*, Vol 4, Number, 114, (2006).

[6] Colton, D., Fife, L., Thompson, A., Building a Computer Program Grader, *Information Systems Education Journal*, Vol 3, Number, 6, (2005).

[7] Coplien, J. O., Multi-Paradigm Design for C++, *Addison-Wesley* (1998).

[8] Harris, J. A., Adams, E. S., Harris, N. L., Making program grading easier: but not totally automatic, *Journal of Computing Sciences in Colleges* Vol 20, Issue 1, (2004) 248–261.

[9] Helmick, M. T., Interface-based Programming Assignments and Automatic Grading of Java Programs, *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, (2007) 63–67.

[10] Hernyák, Z., Király, R., Teaching programming language in grammar schools, *Annales Mathematicae et Informaticae*, Vol 36, (2009) 163–174,

[11] Hext, J. B., Winings, J. W., An automatic grading scheme for simple programming exercises, *Commun. ACM*, 12(5), (1969) 272–275.

[12] Hitchner, L. E., An automatic testing and grading method for a C++ list class, *ACM SIGCSE Bulletin* Vol. 2, Issue 2, (1999) 48–50.

[13] Hitz, M., Kögeler, S., Teaching C++ on the WWW, *Proceedings of the 2nd conference on Integrating technology into computer science education*, (1997) 11–13.

[14] Horwitz, S., Addison-Wesley's Review for the Computer Science AP Exam in C++, *Addison-Wesley* (1999).

[15] Juhász, Z., Juhás, M., Samuelis, L., Szabó, Cs., Teaching Java programming using case studies, *Teaching Mathematics and Computer Science* 6/2, pp. 245–256, 2008

[16] Karlsson, B., Beyond the C++ Standard Library: An Introduction to Boost, *Addison-Wesley Professional* (2005).

[17] Kozma, L., Frohner, Á., Kozsik, T., Porkoláb, Z., Beyond 2000, Beyond Object-Orientation, *Proceedings of 5th International Conference on Applied Informatics*, (2001) 125–134

[18] Meyers, S., Effective STL, 3rd Edition, *Addison-Wesley* (2001).

[19] Nordquist, P., Providing accurate and timely feedback by automatically grading student programming labs, *Journal of Computing Sciences in Colleges* Vol 23, Issue 2, (2007) 16–23

[20] Placer, J., The Promise of Multiparadigm Languages as Pedagogical Tools, *Proceedings of the ACM conference on Comp. Sci.*, (1993) 81–86.

[21] Porkoláb, Z., Zsók, V., Teaching Multiparadigm Programming Based on Object-Oriented Experiences, *Tenth Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts (TLOOC)* (2006).

[22] Saikonnen, R., Malmi, L., Korhonen, A., Fully Automatic Assessment of Programming Exercises, *Proceedings of the 6th annual conference on Innovation and technology in computer science education*, (2001) 133–136.

[23] Stroustrup, B., Learning Standard C++ as a New Language, *C/C++ Users Journal*, (May 1999) 43–54.

[24] Stroustrup, B., The C++ Programming Language, Special Edition, *Addison-Wesley* (2000).

[25] Stroustrup, B., Programming, Principles and Practice Using C++, *Addison-Wesley* (2008).

[26] Tremblay, G., Labonte, E., Semi-automatic marking of java programs using junit, *Proceedings of International Conference on Education and Information Systems: Technologies and Applications (EISTA '03)*, (2003) 42–47.

[27] Westbrook, D. S., A Multiparadigm Language Approach to Teaching Principles of Programming Languages, *29th ASEE/IEEE Frontiers in Education Conference*, (1999) 11b3–14.

[28] Zave, P., A Compositional Approach to Multiparadigm Programming, *IEEE Software VI(5)*, (1989) 15–25.

Norbert Pataki
Zalán Szűgyi
Department of Programming Languages and Compilers
Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
e-mail:
`patakino@elte.hu`
`lupin@ludens.elte.hu`