Годишник на секция "Информатика"
Съюз на учените в България
Том 6, 2013, 111–119

Annual of "Informatics" Section
Union of Scientists in Bulgaria
Volume 6, 2013, 111–119

C++11: C++ AS CONTEMPORARY PROGRAMMING LANGUAGE

Petar Armyanov, Atanas Semerdzhiev, Trifon Trifonov, Magdalina Todorova, Maria Nisheva-Pavlova, Georgi Penchev

Faculty of Mathematics and Informatics, Sofia University
5 James Bourchier Blvd., Sofia 1164, Bulgaria

Abstract: This study presents the main changes to the well-known language C++, introduced in the last version of the C++ standard – C++11. These changes significantly improve the abstraction mechanisms of the language and expand the set of abstractions which can be expressed in C++. They once again bring C++ to the list of popular modern languages. The article presents most of the changes related to improvements of performance and improvements to the way the language is used in practice. It also contains examples of their usage.
Keywords: Programming languages, Modern languages, C++, C++11.

1. History

The C++ programming language was developed in the early 1980s as a general-purpose language with certain bias towards system programming. The main goals were: to improve the C language, to support data abstraction, to support object-oriented programming, to support generic programming [1]. During the next 30 years, the C++ language gained significant popularity and saw rapid development. During its development, the language was standardized several times by ISO/IEC. After 1998, the development of a new standard began, known as 0x, because it was expected to be finished in the early years of the new century. However, the ideas introduced in this new language were ambitious and its development took more time than it was first anticipated. A partial standard was published in 2003 and it fixed a lot of details from the previous standard from 1998 [2]. However, the work on the new capabilities was still ongoing. The goal was to make such changes in C++, as to solve the following problems [1]:

- Make C++ a better language for systems programming and library building – that is, to build directly on C++'s contributions to programming, rather than providing specialized facilities for a particular sub-community (e.g. numeric computation or Windows-style application development).
- Make C++ easier to teach and learn – through increased uniformity, stronger guarantees, and facilities supportive of novices (there will always be more novices than experts).

Thus, after a big delay, the new standard – ISO/IEC 14882:2011, was born. It introduces a wide array of new capabilities, such as lambda expressions, standardized multithreading and synchronization libraries, as well as new features for the object-oriented capabilities of the language [3]. The new capabilities once again bring the language to the group of the most widely used modern languages [4].

In [5], the main changes in the C++11 standard can be found, among which: extensions to the core language; core language runtime performance enhancements; improvements in compilation speed; improvements to the way the language is used in practice; improvements to the core language; changes in the standard library; removal of obsolete features.

For the sake of brevity, in this article we will present only new features, which are related to improvements of performance and improvements to the way the language is used in practice.

2. Some new features of the language

Descriptions of the new constructs of the language can be found in the official documentation and in other sources, such as [6, 7, 8].

2.1. Lambda expressions

Lambda expressions make it possible to define a function locally – at the same place, where the call is made. Thus it is no longer necessary to define functions with simple implementations or classes of functions, which are typically used with higher order functions from the standard library. A lambda expression in C++11 has the following syntax [6]:

```
[capture](parameters)->return-type {body}
```

The use of the square brackets indicates the beginning of a lambda expression to the parser. The capture clause of the declaration is particularly interesting. It specifies which variables from the external context, in which the function is declared, can be used in its body and how. It is possible to access all or some of the variables by value or by reference, to access the this pointer (when the lambda function is declared in a member-function of a class), or all variables may be hidden so that they cannot be accessed in the body of the function. In the latter case the lambda function can only access its parameters and locally defined variables. The parameters, the type of the return value and the body are similar to those of any other function in the language. One has to mind that for void functions, no return value is specified. Here is a simple example, which counts the number of digits in a string, using lambda expressions in the *for_each* function:

```
int main()
{
     char str[] = "Using C++ is cool!";
     int digits = 0; // will be modified by the lambda
     for_each(str, str+sizeof(str), [&digits] (char c) {
         if (isdigit(c))
            digits++;
    });
    cout << digits << endl;
}
```

In this example, the variable digits can be accessed inside the lambda function by reference. The most widely used constructs in a capture expression are [&], [=], [], which respectively specify that the lambda expression has access to all external variables by reference, all external variables by value and access to none of the variables.

As in most functional languages, the lambda expressions are anonymous. However, in order to satisfy the code reuse rule, the lambda expression can be assigned to a variable and used multiple times with a single definition. The next section contains an example, which shows how this works.

2.2. Auto types

C++ is a strongly typed language. This means that every variable should be declared with a specific type and this type cannot be changed while the variable exists. Sometimes this may cause problems – often the type of the variable is more complex than its declaration, e.g. with complex pointers to functions and arrays, iterators or other template classes. The following code gives an example:

```
vector<string>::iterator it = v.begin();
```

In such situations, the type of the variable is known during the initialization, by the type of the value which is being assigned. To make it more convenient to work with such constructs, the new standard introduces the auto type. It requires the variable to be initialized in the declaration and then the type of the variable is deduced from the value, which is being assigned. Thus the above example may be written as:

```
auto it = v.begin();
```

Obviously, this is significantly simpler to write and additionally reduces the probability of introducing errors in the type description and thus also the unwanted castings, which may result from them. Automatic declarations allow the use of modifiers, so we can write:

```
const auto it = v.begin();
```

With automatic types, one can also define arrays of arbitrary types or named lambda expressions:

```
auto sumatorFun = [&sum](int a){sum += a;}
```

Along with auto, a new operator called *decltype* is introduced. It returns the type of a given expression and can be used to declare simple names for complex types – even if they are template-based, i.e. not known at the time of writing the code. For example:

```
const vector<T> v;
typedef decltype (v.begin()) cIt;
for (cIt it = v1.begin(); it != v1.end(); ++it)
```

This new operator and the automatic types make C++ programming simpler and reduce the risk of introducing errors in the code. Also the language gets closer to the contemporary dynamically typed languages.

2.3. Initializations

In the preceding C++ standards, there was no option to initialize dynamic arrays during their declaration. For example, the following expression seems intuitive, but was not valid in standards prior to C++11:

```
int * a = new int[3]{1, 2, 3};
```

The new standard introduces initialization rules, which allow variables of this kind to be initialized with a {} construction. Container classes may also be initialized in this way. Here are some examples:

```
vector<int> numbers = {1, 2, 3};
map<int, string> months = { {1, “January”},
                            {2, “February”}, /* etc.*/ };
```

If the class A has a constructor with two parameters, then the expression A a{0, 3} is valid in C++11 and is equivalent to A a(0, 3) and also to A a = {0, 3}.

Additionally, it is allowed to use {} in the initialization list of a constructor, which assigns initial values to member-arrays:

```
class A
{
        int data[3];
    public:
        A():data{1, 2, 4}{}
};
```

Also it is now possible to specify initial values of non-static members, similar to the currently popular Java and C# languages:

```
class B
{
        int x = 5;
    public:
        B();
};
```

2.4. Rvalue References

In C++ references can only be bound to lvalue expressions and this binding is constant. This may be inconvenient, when a temporary object (e.g. the result of a function call) must be passed in a place which expects a reference. C++ uses the term rvalue for such expressions. Although the definition is more complex, one can think of such expressions as expressions, which can be placed on the right side of an assignment (*operator* =).

Rvalue references are denoted with the *&&* symbol. They can be used to swap two objects by name, i.e. without copying the actual data, by using special copy-constructor and assignment operator with arguments of type rvalue reference. Although this approach may be convenient to improve the performance of different algorithms, which copy or swap elements (e.g. sorting the data in a container), it also introduces certain risks for programmers, which do not fully understand its principles. One of the main issues is that even if a given argument or variable is declared as an rvalue reference, if we use it by its name, it is being treated as an lvalue. This is because the standard follows the principle "If something has a name, it is lvalue". For this reason a special function called std::move is introduced, which translates lvalue to rvalue references and supports the use of move semantics.

2.5. Deleted and defaulted functions. Delegating constructors

The C++ compiler automatically generates certain methods in the classes, even if we do not declare them. Such methods are the copy constructor, the assignment operator, the destructor, sometimes the default constructor and also special operators, such as &. The new standard introduces two new keywords – default and delete.

With default, one can point that a given method will use the default, compiler generated definition, instead of a version written by the programmer. For example:

```
class A
{
    public:
      A() = default;
      A(const A&) = default;
        …
};
```

On first sight such functionality may seem pointless, because if the two lines above were omitted, the same effect would be achieved. However, there are specific differences.

For example, if a class already contains a constructor, the compiler will never generate a default constructor. With the help of the default keyword, one can now quickly use the default implementation in this case. Another application rises from the fact that default may be used not only in the declaration, but also in the implementation of a method. For example if we declare the class A like this:

```
class A
{
    public:
     A();
       …
};
```

then it is possible to define the constructor in the following way:

```
A::A() = default;
```

We should note that default may be specified either in the declaration or in the definition of a method, but not in both places. What makes such an approach convenient is that one can have a class declared in a header file and then create different implementations in one or more *.cpp* files. Some of the implementations may use default, while others do not, depending on the specific needs of each implementation. The use of default is also recommended for improved code readability.

The delete declaration can be used in a wider array of cases. It is used similarly to default, but it has the opposite meaning – a method which is declared as delete is not added to the class and may not be used. For example if we want to prohibit the copying of objects of a class, we may write:

```
class A
{
    public:
      A(const A&)=delete;
       A& operator=(constA&)=delete;
         …
};
```

Similarly to default, delete may be added to the declaration of the method or its implementation.

Another application of this new feature is to prohibit the use of methods with specific parameters. For example, consider that we have a function foo declared as:

```
void foo(int x);
```

This function may be called with an argument of a "lesser" integer type, e.g. char. If we want to prohibit such a call, we may add the following declaration:

```
void foo(char c)=delete;
```

Another new feature pertaining to constructors is the so called delegated constructors. They make it possible to use an already defined constructor in the initialization list of another constructor of the same class. For example:

```
class A
{
    public:
      A(int x, char c) : value(x) {...}
      A():A(3, 'a'){}
        …
};
```

This improves code reuse and is already well-established in languages such as Java and C#.

2.6. Multithreaded programming

There is hardly any doubt that the trend in the development of modern architectures is towards processors with multiple cores. This implies that processes should be able to work with multiple threads. In the earlier versions of C++ there were no standard libraries for multithreading and synchronization. This made it necessary for programmers to use specific third-party libraries for each compiler and/or platform, which contradicts the idea of portability, which is one of the fundamental ideas behind C/C++. In C++11 this problem is solved and the standard library now contains several headers files, which declare functionality similar to that of Boost – the most widely used third party C++ library for this purpose [8]. Only the basic functionality is implemented:

- *Creation of threads*: A new class thread is added. Its constructor receives a pointer to a function or an object of a functional class, which will be executed in the thread. The execution is scheduled to start immediately after the constructor. For example:

```
void thread_fun1()
{
    for (int i = 0; i < 10; ++i)
        cout << i << endl;
}
void thread_fun2()
{
    for (int i = 20; i < 35; ++i)
        cout << i << endl;
}
int main()
{
    thread t1(thread_fun1);
    thread t2(thread_fun2);
    t1.join();
    t2.join();
    return 0;
}
```

- *Synchronization*: The simplest way to synchronize two threads can be accessed through a new class, which implements a standard mutex. The C++11 library also provides mutex with a timeout and recursive mutex. There are also helper classes, such as lock_guard, which implements a simple way to synchronize the execution of a function or

block by locking a mutex in its constructor and unlocks it in its destructor. There are also options for synchronization through a conditional variable. This allows waiting for and notifying other threads.

- *Atomic operations*: In practice, synchronization is most often necessary when accessing memory for simple operations, e.g. incrementing a counter or accessing a boolean condition. The new standard introduces a new class called atomic, which contains an integer value and provides a set of simple operations, which are guaranteed to be atomic.

Although the language does not introduce many options, the selected set provides a quick and hassle-free solution to almost any parallel programming task and, what is most important, the code is completely platform independent and therefore portable.

3. Conclusion

C++11 is a standard for the C++ programming language, which was accepted in 2011. Although the differences between the two previous standards (C++98 and C++03) are so small, that they are often paid little or no attention, the new standard transforms C++ into a new, modern language. In [1] the creator of the language – Bjarne Stroustrup, says: "Surprisingly, C++11 feels like a new language: The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever".

This article describes some of the main changes introduced in C++11, which significantly improve the abstraction mechanisms of the language and expand the set of abstractions which can be expressed in C++ naturally, elegantly, flexibly and effectively. The experience accumulated from applying the new features of the language in the education of students in „Informatics and Computer Sciences" is described in [9, 10, 11]. The analyses made in these articles are optimistic as to the second main goal, which motivated the creation of C++11: „make C++ easier to teach and learn – through increased uniformity, stronger guarantees, and facilities supportive of novices".

Acknowledgements

The presented study has been supported by the Bulgarian National Science Fund within the project titled "Contemporary programming languages, environments and technologies and their application in building up software developers", Grant No. DFNI-I01/12.

References

[1] *C++11 - the new ISO C++ standard*. http://www.stroustrup.com/C++11FAQ.html

[2] *INTERNATIONAL STANDARD ISO IEC 14882:2003-10-15*. Programming languages C++

[3] *INTERNATIONAL STANDARD ISO IEC 14882:2011-09-11*. Programming languages C++

[4] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[5] http://en.wikipedia.org/wiki/C++

[6] Kalev, D., *The Biggest Changes in C++11 (and Why You Should Care)*. 2011 http://blog.smartbear.com/software-quality/bid/167271/The-Biggest-Changes-in-C-11-and-Why-You-Should-Care

[7] Deitel, P., H. Dietel, *C++ 11 for programmers*, Chapter 24, Prentice Hall, 2013.

[8] Williams, A., V. J. Botet Escriba, *Boost C++ Libraries, Chapter 30. Thread 4.0.0,* http://www.boost.org/doc/libs/1_53_0/doc/html/thread.html

[9] Donchev, I., Teaching C++11 at Introductory Level, *Proceedings of the 8th Annual International Workshop on Computer Science and Education in Computer Science (CSECS)*, 5-9 July, 2012, Boston-New York, 7–18.

[10] Дончев, И., Move семантиката в учебния курс по обектно-ориентирано програмиране на C++, *Сборник научни трудове на MATTEX 2010*, 19–20 ноември 2010 г., Шумен, 2011, 380–387.

[11] Donchev, I., Experience in Teaching C++11 within the Undergraduate Informatics Curriculum, *International Science Journal Informatics in Education*, Vilnius University, Vol. 12, 2013, No. 1, 1–21.